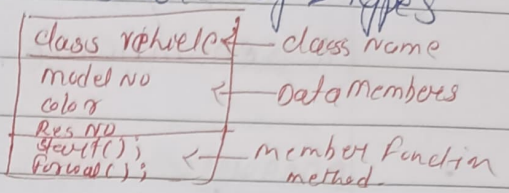
Date: 24/9/24

Class

- A java class is a definition block on used to define the states and behaviors of an object

- A class contains members, members are of 2 types
  (1) Data Member
  (2) Member Function



- Data members are variables used to define the states of an object
- Member Function are methods used to define the behaviors of an object

Object (instance)

An object is a real world entity that has its own states and behaviors.

Syntax: `Classname variable = new Classname();`

- Classname → datatype
- variable → Reference variable
- new → Keyword
- Classname() → Constructor Call

- In Java object is a copy of the class that contains all the non-static members.

- The non-static member in a class gets loaded to memory during object creation.

Reference Variable

- Reference Variable Any variable that is declare using non primitive datatype is called as reference variable.

Date :

- Generally we declared reference variable using following name as datatype

* Classname
* interfacename
* enumname.

Classname : reference variable = object reference / Address of object
null reference / null

- A reference variable in Java can hold either address of the object or null value

```
Class Actor
{
  void acting()
{
    Sopln("Actor is Acting");
}
}
public class MainClass1
{
    Actor a1 = new Actor();
    Actor a2 = null;
    Sopln(a1);
    Sopln(a2);
    a1.acting();
    a2.acting(); // -- exception : nullpointerException.
```

reference

(ek null object par ek method ya operati call karte hai tab)

Date :

Null

- null is a Keyword in Java.
- null meaning is nothing.
- null == 0
- If a reference variable is assigned to null it is called null reference.
- null is the default value for all reference variables.

** - If we try to access any non-static members using a null reference it will cause "NullPointerException".

Shallow Copies :-

```
Class Laptop
{
    String wallpaper = "Deepika.jpeg";
}

public class Mainclass54
{
    public static void main(String
{
    Laptop l1 = new Laptop();
    Laptop l2 = l1;

    Sopln(l1.wallpaper);
    Sopln(l2.wallpaper);
    l1.wallpaper = "Alia.png";
    Sopln(l1.wallpaper);
    Sopln(l2.wallpaper);
```

Date :

Deep Copies and shallow Copies

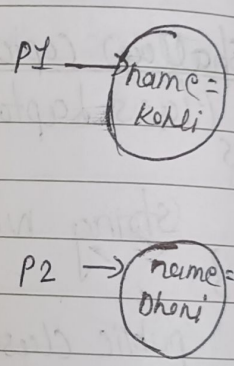
Deep Copies

- Multiple reference variables of the same class with each reference variable assigned to a distinct object is called as Deep Copies

- In Deep Copies changes made by one reference variable will not affect the others

```
class Player
{
    String name = "Dhoni";
}

Player P1 = new Player();
Player P2 = new Player();
Sopln(P1.name); // Dhoni
Sopln(P2.name); // Dhoni
P1.name = "Kohli";
Sopln(P1.name); // Kohli
Sopln(P2.name); // Dhoni
```

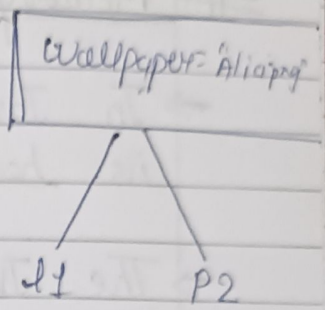


Shallow Copies

- Multiple reference variables of the same class assigned to a common object are called as shallow Copies.
- In Case of shallow Copies changes made by one reference variable will affect the other.

```
class Laptop
{
 String wallpaper = "Deepika.jpeg";
}
Laptop l1 = new Laptop();
Laptop l2 = l1;

Sopln(l1.wallpaper);
Sopln(l2.wallpaper);
l1.wallpaper = "Alia.png";
Sopln(l1.wallpaper);
Sopln(l2.wallpaper);
```



** Interview Question

① What is class?
- Definition
- member, classify
- Data members
- member Function

② What is object
- Definition
- Syntax
- Point-1
- Point-2

③ What is a Reference variable?
- Definition
- Created
- Stored.

④ What is null? When do we get nullPointerException
- null is a keyword
- null meaning is nothing
- When a reference variable is assigned to null it is called null reference
- null is default value for all reference variable
- if we try to access non-static members using null reference it will cause NullPointerException.

Date :

(5) Difference between Deep Copies and Shallow Copies?

25/9/24 :

Memory Management In Java

⇒ In Java entire memory management is handelled by the JVM at runtime

⇒ The JVM makes use of the following memory areas

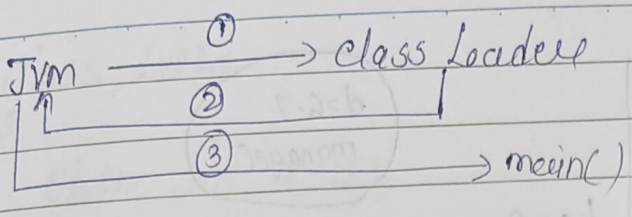
(1) class area
(2) heap area
(3) method area
(4) stack area

Ex:

```
class Demo
{
  static int a = 20;
         int b = 20;
  static void test()
  {
    Sopln("executing test()....");
  }
  void disp()
  {
    Sopln("executing disp()....");
  }
  public static void main(String[] args)
  {
    Demo d = new Demo();
  }
}
```

Date :

JVM —①→ class Loader
②
③ → main()



	non-static b=20 disp()
STACK	Heap
method	class static
{ defn of test } { defn of main } { defn of disp }	class Demo { a=10 → Declaration test() main()

② class sample
{
static char ch='A';
double val =7.8;
static void run()
{ sopln(" ");
}

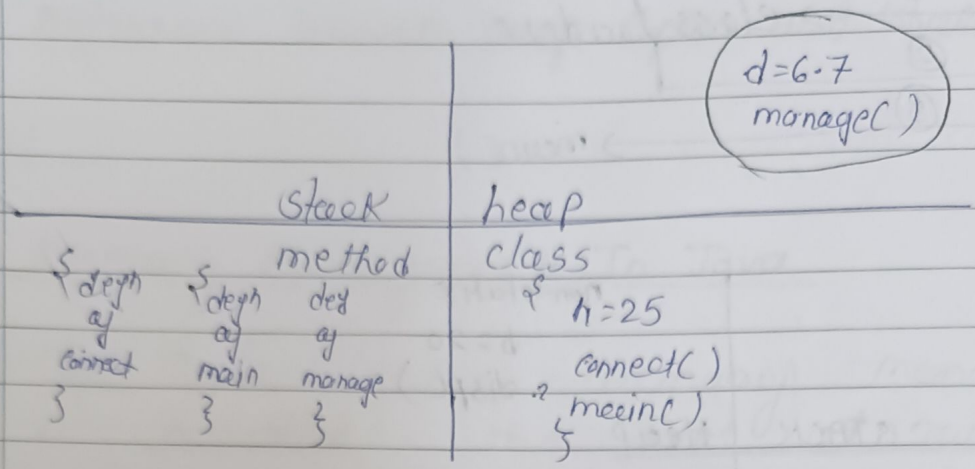
void play()
{ sopln(" ");
}

public static void main(string[] args)
{
sample ref=new sample();
}
}

val=7.8
play()

stack	heap
{ defn of run } { defn of main } { defn of play } method	class class sample { ch=A run() main()

Date :




```
class Example
{
  static int h=25;
  double d=6.7;
  void manage()
{
  sopln
}

static void connect()
{
  sopln
}

Public Static void main(string[] aregs)
{
   Example ex=new Example();
}
}
```

Date :

(3)

```
Class Delta
{
  Static int x=15;
  int y = 25;
  Static void play()
{
  sopln
}
  void test()
{
  sopln
}
public static void main(
{
  Delta d1 = new Delta()
  Delta d2 = new Delta()
}
}
```

Stack				heap
method				class
{degh of Play }	{degh of main }	{degh of test }	{degh of test }	class Delta { x=15 play() main() }

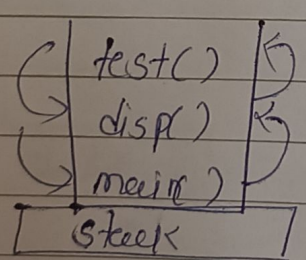
y=25 test()

y=25 test()

Date :

(4)

```
Class Alpha
{
  static void test()
  {
    sopln("test starts");
    sopln("test ends");
  }
  static void disp()
  {
    sopln("disps starts");
    test();
    sopln("disps end");
  }
  public static void main(string[] args)
  {
    sopln("main start");
    disp();
    sopln("main end");
  }
}
```



stack			heap
method			~~mai~~ class
{ defn of test }	{ defn of disp }	{ defn of main }	class Alpha { test() disp() main() }

Date :

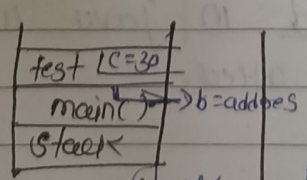
Entry order	Exit order
(1) main( )	test( )
(2) disp( )	disp( )
(3) test( )	main( )

LIFO - Last in First out.

main starts
disp starts
test start
test ends
disp ends
main ends

(5)

```
Class Beta
{
  Static int a=10;
  int b=20;
  void test()
{
    int c=30;
  sopln("executing test()-- ");
}
  public Static void main(string[] args)
{
    Beta b=new Beta()
    b.test();
```



b=20
tes( )

Stack method	heap class
{defn of main }	{def of test }
	class Beta { a=10 main() }

Date :

Class Area

- All static members are loaded to class area.
- Static members are loaded to class area by the class loader.
- Static members are loaded to memory by default during class loading time
- There exist only one copy of static memory members in memory.
- Class area is also known as static pool

Heap area

- The non-static members are loaded to heap area.
- non-static members are loaded to heap area by JVM
- non-static members are loaded to memory during instantiation.
- non-static members multiple copies may be present in heap area.
- All object created in java will be allotted memory in heap area.

Method area

- All method definitions are stored in the method area.

Stack area

- Stack area is also known as execution memory
- All method executions is done in the stack area.
- Stack area follows a LIFO or FILO order of execution.
- All local variables are allotted memory in stack area.

Difference between static and non-static

static	non-static
(1) Created with static keyword	(1) created without static keyword
(2) accessed using classname	(2) accessed by creating an instance
(3) loaded to class area	(3) loaded to heap area
(4) only 1 copy exists in memory	(4) multiple copies may exist in memory
(5) loaded during class loading	(5) loaded during instantiation
(6) loaded by class loader	(6) loaded by JVM
(7) class level member	(7) instance member.

Date : 26/9/24

Initialization Blocks

- These are special block in a java program that are used to initialize the data members present in a class

- Initialization Block are classified into two types

(1) Static initialization Block
(2) non-static initialization Block

(1) Static initialization Block

(i)- Static Blocks are created using static keyword
(ii)- Static blocks are used to initialize static data members
(iii)- In a single static block we can initialize multiple static data member.
(iv)- We can create multiple static blocks inside a class
(v)- static blocks are executed by the class loader at the time of class loading
(vi)- static blocks will execute only once in the program lifetime.

Non-static Initialization Blocks

(1) → Non-static blocks are created without using static keywords
(2) → Non-static blocks are used to initialize non-static data member.

(3) In a single non-static block we can initialize multiple non-static data members.

(4) We can create multiple non-static blocks inside a class.

(5) Non-static blocks are executed by the JVM at the time of object creation.

(6) non-static blocks may execute more than once in a program lifetime.

(7) It is also known as Instance initialization Block.

Note:-

```
class Alpha
{
  static int i;
  {
    i = 20;
  }
  public static void (String[] args)
{
  A
  Sop(Alpha.i)
```

Notes:-> It is possible to initialize static data member inside a non-static block however it is not recommended as the initialization of static member becomes subject-to-object creation.

Date :

Note ↦ It is possible to do any type of coding in initialization Block however But it is recommanded only right the code related to initialization activity.

(Q) Difference between static Block & non static Block?

Static Block	Non-static Block
(1) Created with static Keyword	(1) created without static Keyword
(2) Executed by classloader	(2) Executed by JVM
(3) executed during class loading	(3) executed during object creation.
(4) Executes only once	(4) executes multiple times
(5) Responsible to initialize static data member	(5) Responsible to initialize non-static data member

(Q) What is classLoader?

- It is a sub-system of the JVM present inside the JRE
- It is primarily responsible for,
  (1) Loading all the static members to memory
  (2) Executing all the static initialization block

The time interval in which class loader works is known as class loading time.

(Q) We can initialize non-static data members inside a static block

A NO

(Q) We can initialize static data members inside a instance block?

A- Yes, but it is not recommended.

27/9/24

Constructor

① Constructor are special members in a class that are used to initialize non-static data members

② every class in Java will have a constructor.

③ Constructor will always have the same name as the class name

④ Constructor gets executed during object creation

⑤ In Java Constructor are classified into two categories.

(1) Default - Javac.
(2) zero argument Constructor] - User defined Constructor
(3) parameterized Constructor]

Date :

(Q) We can initialize non-static data members inside a static block

A NO

(Q) We can initialize static data members inside a instance block ?

A. Yes, but it is not recommended.

27|9|24

Constructor

① Constructor are special members in a class that are used to initialize non-static data members

② every class in Java will have a constructor.

③ Constructor will always have the same name as the class name

④ Constructor gets executed during object creation

⑤ In Java Constructor are classified into two categories.

(1) Default - Javac.
(2) zero argument Constructor
(3) parameterized Constructor
] - User defined Constructor

Date :

Ex ①

```
class Demo
{
    int a;
    double b;
    String s;
}
public class Mainclass1
{
    public static void main(String[] args)
    {
        Demo ref = new Demo(); // object is created by calling the default constructor
        Sopln(ref.a); // 0
        Sopln(ref.b); // 0.0
        Sopln(ref.s); // null
    }
}
```

Ex ②

```
class Sample
{
    double val;
    void test();
    {
        Sopln("executing test() ");
    }
}
public class mainclass2
{
    public static void main(String[] args)
    {
        Sample s = new Sample(); // creating object by calling default constructor.
        Sopln(s.val);
        s.test();
    }
```

Date :

Default Constructor

(1) The default Constructor is provided by Javac (compiler)
(2) The compiler provides default Constructor when programmer fails to create a Constructor.
(3) The default Constructor provides default values for the non-static data members. which are not initialized.

Ex(3)

```
Class Alpha
{
    int val;
    Alpha()
    {
        Sopln("executing Constructor");
        val = 25;
    }
}
public class Mainclass3
{
    public static void main(string[] args)
    {
        Alpha at = new Alpha();
        Sopln(at.val);
    }
}
```

Date :

Default Constructor

(1) The default Constructor is provided by Javac (compiler)
(2) The compiler provides default Constructor when programmer fails to create a Constructor.
(3) The default Constructor provides default values for the non-static data members which are not initialized.

Ex(3)

```
Class Alpha
{
    int val;
    Alpha()
    {
        Sopln("executing Constructor");
        val = 25;
    }
}
public class Mainclass3
{
    public static void main(String[] args)
    {
        Alpha a1 = new Alpha();
        Sopln(a1.val);
    }
}
```

Zero. Argument

```
Class Beta
{
    int a;
    double d;
    char c;
    Beta()
    {
        Sopln("executing Beta Constructor");
        a = 15;
        d = 4.5;
        c = 'A';
    }
}
public class Mainclass4
{
    public static void main(String[] Jargs)
    {
        Beta obj = new Beta();
        Sopln(obj.a);
        Sopln(obj.d);
        Sopln(obj.c);
    }
}
```

Zero Argument Constructor.

(1) It is created by the programmer.
(2) It is possible to initialize non-static data members in the body of zero Argument Constructor.
(3) As the name suggests it does not accept any arguments.

parameterized

Date :

Ex1-
```
Class Example
{
    int a;
    Example(int arg)
    {
        Sopln("executing example Constructor");
        a = arg;
    }
}
public class Mainclass5
{
    public
    {
        Example ex1 = new Example(15);
        Sopln(ex1.a);
        Sopln("---------");
        Example ex2 = new Example(25);
        Sopln(ex2.a);
    }
}
```

Ex1-
```
Class Delta
{
    int num;
    double val;
    char ch;
    Delta(int a, double d, char c)
    {
        Sopln("executing Delta Constructor");
        num = a;
        val = d;
        ch = c;
    }
```

```
}
public class mainclass6
{
public static void main(String [] aregs)
{
Delta d=new Delta(10,6.8,'A');
Sopln(d.num);
sopln(d.val);
sopln(d.ch);
}
}
```

Parameterized Constructor

(1) It is created by the programmer.
(2) In parameterized Constructor we can initialize non-static data members with the help of areguments inside the Constructor.
(3) As the name suggests it accepts aregument

Note:-
(1) When the programmer wants to create object for a class but is not concern about initializing the non-static data member than default Constructor is best option.

(2) When the pro want to create object for a class with all object having same initialization for the non-static data member. Zero aregument is best suited.

Date :

(3) When the programmer wants create object and provide different initialization value for the non-static data member each object then parameterized Constructor is best suited.

(8) Can we given return type for a Constructor?
A No, if by mistake return type is given it behaves like a method.

(8) Can Constructor have modifiers?
A No, but we can provide access modifiers. With respect to normal modifiers

(8) Can we initialize static data member inside a Constructor
Yes, but it is not recommended as it is not the Correct approach.

28/9/24

Constructor Overloading

Constructor overloading is the Concept of developing multiple Constructor inside a class

Rule → When creating multiple Constructor in a class it is mandatory to vary the argument list in each Constructor.

We can vary the argument in three ways
(1) Type of argument
(2) length of argument
(3) Sequence of argument

Date :

(3) When the programmer wants create object and provide different initialization value for the non-static data member each object then parameterized Constructor is best suited.

(8) Can we given return type for a Constructor?
A No, if by mistake return type is given it behaves like a method.

(8) Can Constructor have modifiers?
A No, but we can provide access modifiers. With respect to normal modifiers

(8) Can we initialize static data member inside a Constructor
Yes, but it is not recommended as it is not the Correct approach.

28/9/24

Constructor Overloading

Constructor overloading is the Concept of developing multiple Constructor inside a class

Rule → When creating multiple Constructor in a class it is mandatory to vary the argument list in each Constructor.

We can vary the argument in three ways
(1) Type of argument
(2) length of argument
(3) Sequence of argument

Date :

① Type of arguments

```
Class Delta
{
  Delta (int a)
  {

  }
  Delta (double d)
  {

  }
  Delta (String s)
  {
  }
}
```

② Length of argument

```
Class Alpha
{
  Alpha (int a)
  {
  }
  Alpha (int a, int b)
  {
  }
  Alpha (int a, int b, int c)
  {
  }
}
```

Date :

③ Sequence of arguments

```
Class sample
{
  Sample (int a, , string s )
  {

  }
  sample ( strings, int a )
  {

  }
}
```

→ Constructor overloading is used to create object of the same class with different initialization capabilities

Ex)→

```
Class Amazon
{
  Amazon( )
  {
    Soph ("Normal Amazon Account : shopping");
  }
  Amazon(double fees)
  {
    Soplr("Prime Amazon Account : shopping, video, music");
  }
}
Public class Meinclass1 {
public static void main (String [] args)
{
  Amazon a1 = new Amazon( );
  Amazon a2 = new Amazon(999.0);
}
```

Constructor overloading

Date :

Ex 2

```
Class Employee
{
  Employee (double ctc)
  {
    sopln("Engineer")
  }

  Employee (double etc, int bonus)
  {
    sopln("Team lead")
  }

  Employee (double ctc, int bonus, int incentive)
  {
    sopln("manager")
  }
}

publie class Mainclass2
{
public static void main(String[] args)
  Employe e1=new Employee(5.6);
  Employe e2=new Employee(5.6,2);
  Employe e3=new Employee(5.6,4,7);
  }
}
```

Constructor overloading

Date :

Ex 3

```
Class IPhone16
{
 IPhone16( )
{
  Sopln (" IPhone16 ") ;
}

IPhone16( int a )
{
 Sopln("  IPhone16 plus ") ;

}
IPhone16( double d )
{
 Sopln("  IPhone16 plus Pro ") ;

}
IPhone16( String s )
{
  Sopln (" IPhone16 Pro max ") ;

}
}

public class Mainclass3
{
public static void main( String [] args )
  IPhone16 I1 = new IPhone16 ( ) ;
  ——— I2 = ——— (10) ;
  ——— I3 ——— (5.6) ;
  ——— I4 ——— ("hi") ;
```

Date :

```
Class Antivirus
{
  Antivirus()
  {
    Sopln("basic scan");
  }

  Antivirus(double d)
  {
  Sopln("full scan");
  }

  Antivirus(String s)
  {
    Sopln("full scan server scan");
  }
}

public class mainclass4
Antivirus A1=new Antivirus();
          A2=new Antivirus(5.8);
          A3=new Antivirus("hello");
  }
}
```

Date :

Eclipse

→ Eclipse is an (Integrated Development Environment)
→ It is a free software distributed by Eclipse Foundation.
→ It allows programmers to write, Compile, execute and debug java programs from a single window

Advantages of eclipse

→ It provides shortcuts that can save program time and effort
→ It auto organizes all the imports.
→ It allows easy integration Support for 3rd Party Libraries.
→ provides plugins for accessing external applications like browsers, web-servers etc.

30/9/24

Passing And Returning Object

① Public class Employee {
    int eid;
    double ctc;
    Employee(int a, double d) {
        eid = a;
        ctc = d;
    }
}

public class EmployeeUtility {
    Static void displayEmployeeDetails(Employee e)
{
    Sopln("Eid: " + e.eid);
    sopln("Eid: " + e.ctc);
}

Date :

```
static Employee createEmployee() {
  Scanner scn = new Scanner(System.in)
  System.out.println("enter Eid");
  int id = scn.nextInt();
  System.out.println("enter ctc");
  double c = scn.nextDouble();
  Employee emp = new Employee(id, c);
      return emp;
  }
}

Public class MainClass1 {
  p s v m (—)
  {
    Employee e1 = EmployeeUtility.createEmployee();
    EmployeeUtility.displayEmployeeDetails(e1);
    sopln("_____");

    Employee e2 = EmployeeUtility.createEmployee();
    EmployeeUtility.displayEmployeeDetails(e2);

    Employee e3 = EmployeeUtility.createEmployee();
    EmployeeUtility.displayEmployeeDetails(e3);
```

② public class circle {

```
    double radius;
    circle(double radius)
    {
        this.radius = radius;
    }
}
```

Date :

```
Public class circleManager{
 static circle createcircle()
{
 Scanner scn=new Scanner(System.in);
  Sopln(" enter radius");
 double rad=scn.nextInt();
 circle cir=new circle(rad);
  return cir;
}
 static void displayCircle(circle c)
 {
   Sopln(" circle radius:"+c.radius);
 }
}
 public class Mainclass3{
  p s v m (--){
   circle cr = circleManager.createcircle();
   circleManager.displaycircle(cr);
  }
}
```

③

```
class Product{
   int pid;
   double price;
  product(int pid, double price){
      this.paid = pid;
      this.price = price;
  }
}
 class productHelper{
 static product createproduct(){
 Scanner scn=new Scanner(System.in);
  Sopln("enter the pid");
  int id=scn.nextInt();
  Sopln("enter the price");
```

Date :

```
double pr = scn.nextDouble();
product p = new product(id, pr);
    return p;
}
static void displayProductinfo(Product p) {
    sopln("pid" + p.pid);
    sopln("price" + p.price);
}
public class mainclass {
    psvm() {
        productHelper1 = productHelper.createProduct();
        productHelper.displayProduct info(p1);
        Product p2 = ProductHelper.createProduct();
        productHelper.displayProduct info(p2);
    }
}
```

Date : 1/10/24

Inheritance

- Inheritance is the Concept of a class and acquiring ~~quering~~ a properties of an unother class

General 2 Points of Inheritance I,

1- Inheritance Can be done using extends keyword

2- The Concept of Inheritance is also known as Is- A- Relationship

Ex:-

```
Class A
{
    - Super
    - Base
    - parent

}
class B
    - sub
    - derived
    - child
}
```

Date :

ex①

```
Class Alpha{

    int a=10;
    void play(){
      Sopln("                    ");
    }
}

class Beta extends Alpha{
      int b=20;
      void send(){
      Sopln("                    ");
    }
}

public class meinclass2{

public static void main(String[]args)
{
    Beta ref=new Beta();
    Sopln(ref.a);
    ref.play();

    Sopln(ref.b);
    ref.send();
  }
}
```

Date :

Rules of Inheritance

(1) The sub class not Acquire the following members of the supere class.

(a) Poivate members
(b) static members
(c) Constructors
(d) Initialization Blocks.

(2) The supere class can not access properties of the sub class

(3) when a class is declared as final it can not have a sub class

Advantages of Inheritance

(1) Code Reusability
(2) S/W Enhancement
(3) Code modification (method overriding)
(4) Generalization vs specialization.

Date :

Ex ①

```
class Instagramold
{
    void post()
    {
        sopln("upload photos");
    }
}
class InstagramNew extends Instagramold
{
    void messenger()
    {
        sopln("send Message");
    }
}

public class Mainclass3
{
public static void main(String[] args)
    Instagramold i1 = new Instagramold();
        i1.post();

    Instagramnew i2 = new Instagramnew();
        i2.post();
        i2.messenger();
}
}
```

Date :

Ex ②

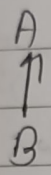
```
class whatsapp1
{
    void message()
    {
        sopln("send/Receive message");
    }
}
class whatsapp2 extends whatsapp1
{
    void call()
    {
        sopln("Audio/video call");
    }
}
class whatsapp3 extends whatsapp2
{
    void status()
    {
        sopln("show Temp photos");
    }
}
public class Mainclass4
{
    public static
    {
        whatsapp1 w1 = new whatsapp1();
        w1.message();
        whatsapp2 w2 = new whatsapp2();
        w2.message();
        w2.call();
        whatsapp3 w3 = new whatsapp3();
        w3.message();
        w3.call();
        w3.status();
    }
}
```

Date : 2/10/24

Types of Inheritance

(1) Single level Inheritance

In this form of Inheritance the properties of super class will be inherited by only one of class



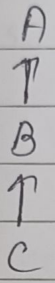
ex -

```
class A
{
  void test()
  {
    System.out.println("executing test()---");
  }
}
class B extends A
{
  void disp()
  {
    System.out.println("executing disp()---");
  }
}
public class mainclass1
{
  P S V m(--)
  {
    A a1 = new A();
    a1.test();
    B b1 = new B();
    b1.test();
    b1.disp();
  }
}
```

Date :

(II) Multi level - Inheritance

In this from of Inheritance the properties of a super class is derived by more than one sub class in different levels



Ex
```
class A
{ void test( ){
    sopln
    }
  }
class B extends A
{
  void disp( ){
      sopln(
    }
  }
class C extend B
{
  void Play( )
{
  sopln
  }
}
Public class meinclass2
{
  P.S.V.m (— )
{
```

Date :

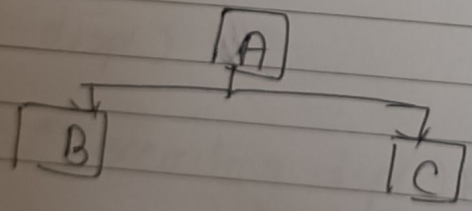
```
A ay = new A();
ay.test();

B by = new B();
by.test();
by.disp();

C cy = new C();
cy.test();
cy.disp();
cy.play();
}
}
```

(III) Hierarchical Inheritance

- In this form of Inheritance the properties of a super class is derived by more than one sub class in the same level.



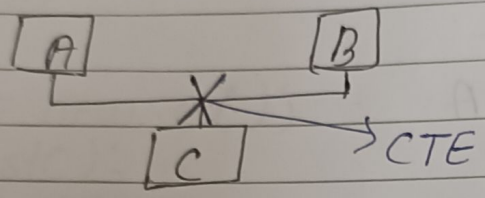

```
Class A
{
   void test()
   {
     Sopln(
   }
}
```

```
class B extends A
{
    void disp();
    {
      Sopln
    }
}
class C extends A
{
    void play()
    {
      Sopln
    }
}
public
{
  P
{
  A a1 = new A();
  a1.test();
  B b1 = new B();
  b1.test();
  b1.disp();
  C c1 = new C();
  c1.test();
  c1.play();
}
}
```

Date :

(iv) Multiple Inheritance

- In this form of Inheritance a single sub class tries to directly Inherit more than one super class. the properties of sup



- This form of inheritance is illegal and not supported in Java. using classes
- multiple inheritance is not supported with classes as it leads to Jacobi's Diamond Ambiguity Problem.
- In multiple Inheritance the sub class doesn't know from which super class it must acquire the properties of object class.
- Multiple Inheritance also create problem with respect to Constructor chaining

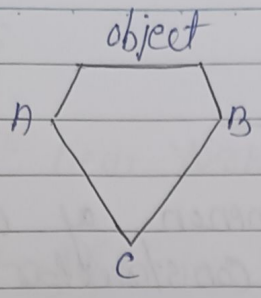
```
Class object
{
}

Class A(object)
{
}

Class B(object)
{
}

Class C extends A, B  CTE
{
}
```

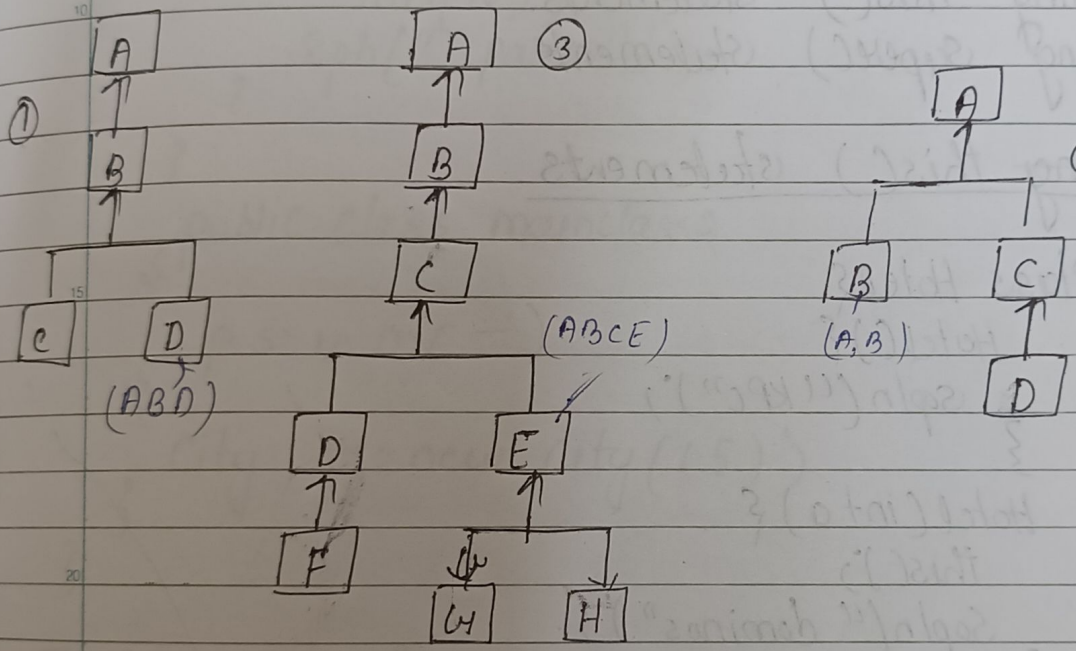
Date :



Jacob's Dimond Ambiguity Problem

Hybrid Inheritance

→ hybrid Inheritance is a Combination of more than one form of inheritance.



(a) What is Generalization and Specialization?

Generalization is the process of developing common functionalities in super class and specialization is the process of developing specific functionalities in sub class

Date :

Constructor Chaining

- It is the phenomenon of one Constructor calling another constructor.
- In Java Constructor chaining can be achived in two ways.

(i) Using this() statements.
(ii) Using super() statements.

Using this() statements

Ex (1)

```
Class Hotel{
    Hotel(){
        sopln("KFC");
    }
    Hotel(int a){
        this();
        Sopln("dominos");
    }
    Hotel(string s){
        this(10);
        Sopln("MC");
    }
}

public class mainclass1{
    s v m (—)
{
    Hotel h1 = new Hotel("ABC");
}
}
```

Date :

Ex(2)

```
Class city{
    city(){
      sopln(" new York");
    }
    city(int a){
        this();
        sopln(" London");
    }
    city(double d){
        this(10);
        soph(" paris");
    }
}
public class mainclass2
{
    p s v m(—)
{
    city cy=new city(1.5);
}
}
```

Ex(3)

```
food class food{
        food(int a){
            this(4.5);
            soph(" Biryani");
        }
        food(double d){
            this("ABC");
            sopln(" masalaDosa");
        }
        food(string s){
            sopln(" pongal");
    }
}
```

Date :

```
Public class mainclass3
{
  p s v m (—)
  {
    food f1=new food(10);
  }
}
```

④
```
Class Amazon
{
  Amazon(){
    sopln("Code to initialize shopping");
  }
  Amazon(double fees){
    this();
    sopln("Code to initialize Prime features");
  }
}

public class Mainclass4{
  p s v m (—)
  {
    Amazon a1=new Amazon(1499.0);
  }
```

This() statements

→ This() statements is used to call another Constructor of the same class.

→ this() we have to write this() statement in the first line of the Constructor.

→ We can only write one this() statement in the Constructor.

→ this() statement can only be written by the programmer.

3/10/24

super() statement

```
- class Demo
  {
    Demo() {
        Sopln("Demo() Constructor");
    }
  }
  class Sample extends Demo {
      Sample() { super();
          Sopln("Sample() Constructor");
    }
  }
  public class Mainclass5 {
    public s v m(—)
  {
      Sample s = new sample();
  }
}
```

Date :

ex (2)

```
class Alpha {

    Alpha() {
      Sopln("Alpha() Constructor");
    }
    Alpha(int a)
    {
      Sopln("Alpha(int) Constructor");
    }
}

Class Beta extends Alpha {

    Beta() {
      Sopln("Beta() Constructor");

    }
}

public class MainClass6
{
public static void main(String[] args)
{
    Beta b = new Beta();

    }
}
```

(3)

③

```
class Delta
{
  Delta()
  {
    this(10);
    Sopln("Delta() Constructor");
  }
  Delta(int a)
  {
    Soph("Delta(int) Constructor");
  }
}
Class Example extends Delta
{
  Example(){
    Sopln("example() Constructor");
  }
  Example(int a){
    this();
    Sopln("example(int) Constructor");
  }
}
public class mainclass7
{
  p s v m (—)
  {
    Example ex=new Example(15);
  }
}
```

Date :

(1)

```
Class Demo
{

}

Class Sample extends Demo
{
    Sample()
    {
        Super(); //
        Sopln("Sample() Constructor");
    }
}
```

Super() // it is calling the default Constructor of the SuperClass

(1)

```
Class Foxtrot
{
    Foxtrot()
    {
        Super(); // it is calling the Constructor of object class
        Sopln(" Foxtrot() Constructor");
    }
}
```

Date :

Note:- When the super class is having only parametrized constructor it is mandatory for the proggramer to create a constructor in the subclass and mannualy write super statement.

(1)

```
class A
{
  A(int a)
  {
  }
}
class B extends A
{
  B(){
  Super(10);
  }
}
```

(2)

```
class A
{
  A()
  {
  }
  A(int x){
  }
}
class B extends A
{

}
```

(3)

```
class A
{
A(int x)
{
}
A(double y)
{
}
}
class B extends A
{
  B()
{
  Super(4.5);
}
}
```

Date :

- In the below to examples a cycle is detected in Constructor calling.
- Whenever there is a cycle in Constructor calling we will get compile time error.

(1)
```
class sample
{
  sample(int a){
  {
    this(7.8);   CTE
  }
  sample(double a)
  {
    this(10);
  }
}
```

(2)
```
class Tiger
{
  Tiger(){
  {
    this();   CTE
  }
}
```

Super() Statement

- Super Statement can be used to called another Constructor of the super class
- Super() Statement should always be written in the first line of the Constructor.
- We can only write one super Statement inside a Constructor.
- The super() Statement can we return either by the programmer or by the Compiler.

Date :

~~class A~~

- Constructor chaining problem associated with multiple Inheritance

Ex.

```
class A                    class B
{                          {

}                          }

class C extends A, B
{
    C()
    {
        super();
    }
}
```

- super() statement in the sub class constructor desn't no which constructor should be called

(Q) what is chaining

(Q) Diff betn this() and super()

Date : 4/10/24

Type Casting

Type casting assigning a value of one datatype to the variable of an another datatype

Type Casting is of two types
(2) non-primitive type Casting
(1) Primitive type Casting
It is the concept of assiging a value of one primitive datatype to the variable of an another primitive datatype.

primitive type Casting is classified into two type
(1) widening
(2) Narrowing

(1) Widening

- It is the Concept of assiging a lower primitive datatype to higher primitive datatype
- Widening is an implicite process that is supported by the ~~Complex~~ Compiler

byte → short → int → long → float → double
char → int

→ Widening

Q

```
public class programm1 {
  psvm(--)
{
  int a =10;
  float f = a; // widening int → float
  sopln(f);

  long val = 1234567l;
    double num = val; // widening long → double
  sopln(num);

  char c = 'A';
  int b = c // widening char → int
  sopln(b);

}
}
```

o/p = 10.0
1234567.0
65

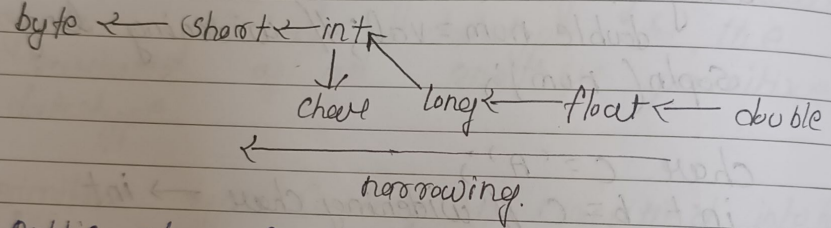
(2)

```
Public class Programm2
{
  Static void run(double d)
{
  Sopln("value:"+d);
}
Public Static Void main(--)
{
  run(4.5);
  run(10); // widening → int to double
  run('A'); // char widening → char to double
}}
```

Date :

Narrowing

- It is the concept of assigning higher primitive datatype to lower datatype
- Narrowing is an explicite process that is ~~on~~ not supported by the Compiler.
- To do narrowing the programmer has to write a Casting stmt



Ex:-
```
Public class Program3
  P S V m (—>)
  double d = 3.4;
  int a = (int) d; // Narrowing :- double —> int
  Sopln (a);

  int x = 97;
  char c = (char) x; // Narrowing : int —> char
  Sopln (c);

  double val = 5.678;
  float num = (float) val; // Narrowing : double —> float
  Sopln (num);
}
}
```

Note:- Boolean does not participate in both widening and narrowing.

True are Keyword.

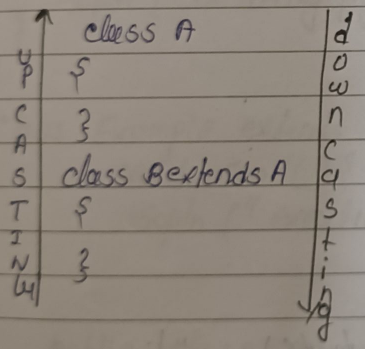
Non-Primitive Type Casting

- It is the concept of assiging the reference varible or object of one class/one type to the refernce variable of another class or type

- To perform non-primitive type casting inheritance is a pre-requisite

- Non-primitive type casting is classified into two type.

(1) up casting
(2) down casting



Up Casting (sub to super)

- It is the concept of assigning a reference variable or object of the sub class to the reference variable of super class.

- Up casting is an implicite process that is supported by the compiler.

Date :

- After upcasting we cannot access the properties of subclass

ex-

```
class Demo
{

}
class sample extends Demo
{

}
public class mainclass1 {
    psvm(-){
  sample obj=new sample();
  Demo ref : obj ; // upcasting : sample --> Demo
  sopln (obj);
  sopln (ref);
}
}
```

(2)

```
class Alpha {
    int a=10;
    void test(){
      sopln(" executing test()---");
  }
}
class Beta extends Alpha {
        int b=20
        void disp(){
        sopln (" executing disp()---");
  }
}
```

```
public class mainclass2{
    psvm(--){
  Beta ref = new Beta();
   sopln(ref.a);
   ref.test();
    sopln(ref.b);
    ref.disp();

 Alpha obj = ref; // upcasting Beta to Alpha
   sopln(obj.a);
   obj.test();
  }
}
```

③
```
class Delta{
 void play(){
   sopln("executing play() ---");
  }
}
class Example extends Delta{
      void send(){
       sopln("executing send() ---");
}
}

  public class mainclass3{
     psvm(--){
  Delta ref = new Example(); // upcasting: Example → Delta
   ref.play();
}
}
```

Date :

(4)

```
Class A
{
  void test()
{
  sopln(" test()----");
}
}

Class B extends A
{
  void disp()
{
  sopln(" disp()----");
}
}

Class C extends B
{
  void play()
{
  sopln(" play()----");
}
}

public class mainclass4 {
  psvm(—) {
    c c1 = new c();
    c1 . test();
    c1 . disp();
    c1 . play();
    b1 = c1 ; // upcasting : C ⟶ B
    b1 . test();
    b1 . disp();
```

Date :

```
A a1 = b1 ; // upcasting: B → A
a1.test();
```

internally discovered B

```
a2 = new C(); //upcasting: C → (B) → A
a2.test();
```

7/10/24

1)
```
public class Fruit{
}
class Apple extends Fruit{
}
class Mango extends Fruit{
}
class Grape extends Fruit{
}

public class FruitBasket{
static void addtoBasket(Fruit obj)
{
    if(obj1 = null)
    {
        System.out.println("Fruit added");
    }
}
}
public class mainclass5{
public static void main(--){
Apple a = new Apple();
FruitBasket.addtoBasket(a);
Mango m = new Mango();
FruitBasket.addtoBasket(m);
Grape g = new Grape();
FruitBasket.addtoBasket(g);
}
}
```

Date :

(7)

```
class Element
{
}
class Button extends Element{
}
class Hyperlink extends Element{
}
class TextBox extends Element{
}

public class TextScript{
 static void textElement (Element e)
   if(e != null)
   {
   Soph ("Add to element");
   }
}
public class mainclass:
{
 p s v m (—){

 Button b = new Button();
 Textscript. textElement(b);

 Hyperlink h = new Hyperlink();
 Textscript. textElement (h);

 TextBox t = new TextBox();
 Textscript. textElement(t);
}
}
```

Date :

-> upcasting is the concept of assigning the object or reference variable of the subclass to the refereene variable of the superclass

-> After upcasting we can not access the properties of the subclass

-> Upcasting is implicite process that is supported by the compilere.

4/10/24

Down Casting (reverese) this is implict process (super to sub)

- DownCasting is the concept of assigning or refereence variable of the super class to the reference variable of the subclass

- DownCasting is a explicite process not supported by the Compiler.

- To perform downcasting programmere has to write Casting statement

- It is mandatory to perform upcasting before downCasting.

- If we perform downcasting without upcasting it will cause "ClassCastException"

- After downCasting we can access the properties of both sub class and superclass

ClassCastException

```
Demo rey = new Demo();
Sample obj = (Sample) rey;
```

obj

Sample — Demo

Sub — Super

DownCasting

```
class Demo
{

}
Class Sample extends Demo
{

}
```

Ex:-

```
Class Demo
{

}
class sample extends Demo
{
}
public class mainclass1{
    p s v m (-) {

        Demo rey = new sample();
        Sample obj = (Sample) rey; // DownCasting
        Sopln(rey);
        Sopln(obj);
    }
}
```

upcasting

Casting statement

Ex

```
class Delta {
    void run() {
        sopln("executing run() -- ");
    }
}
class Example extends Delta {
    void send() {
        sopln("executing send() -- ");
    }
}
public class mainclass2 {
    p s v m(-) {

    Delta d = new Example(); // upcasting
    d.run();
    Example e = (Example) d; // downcasting
    e.run();
    e.send();
}
}
```

→ Casting statements.

Date :

Ex

```
Class A
{
    void test(){
    sopln("test()--");
    }
}

class B extends A
{
    void disp(){
    sopln("disp()--");
    }
}

class C extends B
{
    void play(){
    sopln("play()--");
    }
}

public class mainclass3 {
    ps vm (—)
{
A a1 = new C(); // upcasting : C --B--> A
a1.test();

B b1 = (B) a1; // DownCasting (A --> B)
b1.test();
b1.disp();

C c1 = (C) b1; // Downcasting (B --> C)
c1.test();
c1.disp();
c1.play();
```

```
C c2 = (C) ay; // downcasting (A → C)
c2.test();
c2.disp();
c2.play();

A ay = new B(); // upcasting (B → A)
ay.test();

B by = (B) ay; // Downcasting (A → B)
by.test();
by.disp();

C cy = (C) by; // Downcasting (B → C)
```

Exception : ClassCastException

Instanceof

→ Using Instanceof Keyword we can determine which instance is present in our reference variable.

```
Eg- class Vegetables
    {
    class potato extends vegetable
    {
        Sopln
    }
    class coriander extends vegetable
    {

    }
```

Date :

```
class Lemon extends vegetable
}
}

class zepto{

static void buyvegetable(vegetable v)
{
if(v instancey potato )
{
    sopln(" potato in Kg");
}

else if(v instance cy coriander )
{
    sopln(" coriander in bundles");
}

else if(v instanceay Lemon )
{
    sopln(" Lemon in Pieces");
}
}

class mainclass7{
  PS vm(—){

zepto.buyvegetable(new potato());
zepto.buyvegetable(new coriander());
zepto.buyvegetable(new Lemon());
}
}
}
```

```
        qa.testing();
    }
}
}
public class mainclass1
{
p s v m (--)
{
manager.assignTask(new SoftwareDeveloper());
Manager.assignTask(new TestEngineer());
}
}
```

9/10/24

Method overloading

⇒ Method overloading is the Concept of developing multiple method in a class with same name and different argument list

- We can vary the argument in three ways

(I) Type of arguments
(II) Length of arguments
(III) Sequence of arguments.

- Method overloading is used to perform the same task with different arguments
- method overloading is used to ~~achieve~~ achieve compile time polymorphisms.

- We can overload static, private and final method.

Date :

Ex(I) Type of argument

```
class Example
{
  void play (int a)
  {
  }
  void play (double d)
  {
  }
  void play (string s)
  {

  }
```

(II) Length of Argument

Ex

```
Class Demo
{
  void eat (int a)
  {
  }
  void eat (int a, int b)
  {
  }
  void eat (int a, in b, int c)
  {

  }
}
```

Date :

(iii) Sequence of Arguments

Ex1-

```
Class sample
{
 void book(int a, String s)
 {

 }
void book(String s, int a)
{
}
}
```

① Ex

```
Class FlipKaret
{
  void payment()
  {
  Sopln("COD");
  }
  void payment(long Caed)
  {
    Sopln("Caed");
  }
  void payment(String upi)
  {
   Sopln("UPI");
  }

  void payment(String un, String pwd)
  {
   Sopln("Netbanking");
  }
}
```

```
public class mainclass1
{
p s v m(—)
{
  FlipKaret obj = new FlipKaret();
   obj. payment();
   obj. payment(4354567891011126L);
   obj. payment("pavan@ybl");
   obj. payment("pavan", "pavan123");
}
}
```

Q5)-

```
Class Irctc
{
  void search(int trainNumber)
  {
    Sopln("search with train no ");
  }
  void search(String src, String dest)
  {
    Sopln("search with dest ");
  }
  void search(long pnr)
  {
    Sopln("search with pnr");
  }
}
public main class2 {
  p s v m(—){
  Irctc rey = new Irctc();
  rey. search(12064);
  rey. search("Kar", "bpl");
  rey. search(12346878L);
}}
```

Date :

③ Ex

```
class Facebook
{
  void login(String email, String pwd)
  {
    Sopln("                    ");
  }
  Void login(long mobile, String pwd)
  {
    Sopln("                    ");
  }
}

Public class Meainclass 3
{
  psvm(--)
  {
    facebook fb = new facebook2();
    Fb.login("pavan@gimail.com", "pavan123");
    Fb.login(2834567810L, "pavan123");
  }
}
```

④ Ex

```
class Script {
  void test(int a){
    Sopln("Ad-Hoc-testing");
  }
  Void test(double d){
    Sopln("Smoke testing");
  }
  void test(String s){
    Sopln("Regression Testing");
  }
```

Ex ⑤

```
public mainclass4
{
 p s v m (- )
{
 Script s = new Script();
 s.test(100);
 s.test(3.4);
 s.test("ABC");
}
}
```

Ex⑤

```
Class Mobile
{
 void unlock()
{
 Sopln("                    ");
}
 void unlock(~~pin~~ int pin){
  Sopln("                    ");
  }
 void unlock(Face id){
  Sopln("                    ");
 }
 void unlock(Pattern p){
  Sopln("                    ");
 }
 void unlock(Fingerprint arg)
 {
  Sopln("                    ");
 }
}
```

Date :

```
class Whatsapp
{
    void send(String s){
    Sopln("                ");
    }
    void send(Image a){
    Sopln("                ");
    }
    void send(video v){

    }
    void send(Audio a){
    Sopln(
    }
    void send(Document a)
    {
    Sopln("                ");
    }
}
```

Ex1

```
class Demo
{
    Static void disp();
    {

    }
    Static void disp(int a);
    {

    }
}
```

Date :

Ex
```
class sample
{
private void run()
{
}
private void run(int a)
{
}
}
```

Ex/-
```
class Example
{
final void help()
{
}
final void help(int a)
{

}
```

Ex/-
```
class Delta
{
  void test()
  {
  }
  int test()
  {
    return 10;
  }
}
Delta d = new Delta();
  d.test();
```

CTE

Date : 14/10/24

Method Overriding

①
```
Class Demo {
void test() {
    Soptln (" muncual testing");
  }
}

Class sample extends Demo{
   @Override
   void test() {
     Soptln (" Automation testing");
  }
}

public class Mainclass1 {
p s v m (—) {

   Demo d=new Demo();
   d.test();
  sample s=new sample();
   s.test();
   Demo ref=new Sample(); // upcasting
   ref.test();
  }
}
```

②
```
class Parent {
 void marriage() {
    Soptln (" Arranged marriage"); }
  }
class child extends Parent {
  @Override
 void marriage() {
   Soptln (" Love Marriage");
}
}
```

Date :

```
public class Mainclass2 {
    p s v m (-){
    parent p= new parent();
    p. Marriage();
  child  c=new child ();
    c. Marriage();
    parent rey=new child(); // upcasting
    rey. Marriage();
  }
}
```

o/p
Arrange Marriage
Love marriage
Love marriage

③

```
class father{
  void motorcycle(){
    Soptln("normal splendere");
  }}
class son extends father{
    @overreide
    void motorcycle(){
      soptln("modified splendere");
  }
}
```

```
class  mainclass3{
    ps vm (-){

  father f=new father();
  f. motorcycle();
  Son s=new Son();
  s. motorcycle();
  Father rey=new Son(); // upcasting
    rey. motorcycle();
  }
}
```

o/p
normal splendey
modified splendey
modified splendey

④

```
class FacebookOld{
void reaction(){
System.out.println("Like");
}
}
class FacebookNew extends FacebookOld
{
@Override
void reaction(){
System.out.println("Like Wow Sad Angry haha Love Care");
}
}
public class mainclass45
ps vm(-){

FacebookOld fbold=new FacebookOld();
fbold.reaction();
FacebookNew fbNew=new FacebookNew();
fbNew.reaction();
FacebookOld ref=new FacebookNew();
ref.reaction();
}
}
```

output

Like

Like Wow Sad Angry haha love Care

Like Wow Sad Angry haha love Care

⑤

```
class Whatsapp1 {
    void deliveryReport() {
    sopln("sent")
    }
}
class Whatsapp2 extends Whatsapp1 {
    @override
    void deliveryReport() {
    sopln("sent - delivered")
    }
}
class Whatsapp3 extends Whatsapp2 {
    @override
    void deliveryReport() {
    sopln("sent - delivered - seen")
    }
}

public class mainclass5 {
    p s v m(-)
{
Whatsapp1 =
```

Date :

(6)

```
class GmailV1
{
    void compose()
    {
        sopln("open new tab in Browser");
    }
}
Class extends Gmail V2 extends GmailV1
{
    @override
    void compose()
    {
        sopln("open pop up Box");
    }
}
public class mainclass6
{
    ps v m(-)

    GmailV1 v1 = new GmailV1();
    v1.compose();

    GmailV2 v2 = new GmailV2();
    v2.compose();

    GmailV1 ref = new GmailV2();
    ref.compose();
    }
}
```

output

open new tab in Browser
open pop up Box
open pop up Box

Date :

Method Overreiding

→ It is the process of sub class inhereiting the method from the supereclass and changing the method definition while keeping the same method signature

→ To perform method overreiding inhereitance is a mandatory.

→ We can use method overreiding to upgrade existing software features.

→ We can not overreide static, private and final methods

→ method overreiding is used to achieve runtime polymorphism.

(1) After overreiding and upcasting we will always get latest implementation

(2) Writing @override before method overreiding is optional but highly recommanded

→ @overreide - Annotation - it is used to inform java runtime this a modified version of a method which is present in superClass

Date : 15/10/24

Has-A-Relationship

- It is a form of association in which one class will act as a container for the object of an other class
- there are two forms of has-A-relationship
  (1) Composition
  (2) Aggregation
- The main use of has-A relationship is code reusability.

Ex-①

```
class Demo{
    void test(){
        Sopln("executing test()---");
    }
}
class Sample{
    Demo rey = new Demo();
}
public class mainclass1{
    ps v m(-){
```

Output

```
    Sample obj = new Sample();
    obj.rey.test();
    }
}
```

O/P - Executing test()---

Ex2

```
class Engine
{
    void power(){
    sopln("Engin is Giving power");
    }
}
class Driver{
    void drive{
    sopln("Driver is Driveing");
    }
}
class Car{
Engin eng = new Engin(); // Composition: Car has-a Engine
Driver dr = new Driver(); // Aggregation: Car has-a Driver
}
public class mainclass2{
    p s v m (--)
{
    Car cr = new Car();
    cr. eng. power();
    cr. dr. drive();
}
}
```

Ex3

```
class Camera{
    void takePhoto(){
    sopln("photo clicked");
    }
}
class Airpods{
    Void listen(){
    sopln("listening music");
    }
}
```

Date :

```
class Mobile
{                                    // mobile has-A camera → Composition
 Camera cam = new Camera();//
 Airpods ap = new Airpods();//
}                                    mobile has A Airpods - Aggregation

public class mainclass3 {
   p s v m(-){
mobile mob = new mobile();
mob.cam.takePhoto();
mob.ap.listen();
}
}
```

Ex 4

```
class Teacher{
   void Teach(){
   Sopln("Teaching");
  }
}

 class Department{
 Department
 Teacher tr = new Teacher();
}
class College{
Department dr = new Department();
}

 mainclass4{
   p s v m(-){
College col = new College();
col.dr.tr.teach();
}
}
```

Date :

```
class Printer{
    void write(){
    sopln("executing write()---");
}
void write(int a){
    sopln("executing write(int)---");
}
}

class Computer{
    static Printer out=new Printer();
}

mainclass5 {
ps vm(--){

Computer.out.write();
Computer.out.write(10);
}
}
```

①
```
Package java.io;
public class Printstream
{
    void println()
{

}
.
.
.
}
```

System.out.println();

- System → Inbuilt class in Java Library
- out → static reference variable of Printstream
- println() → non-static method of Printstream

```
package java.lang;
import java.io.Printstream
public class System
{
static Printstream out= new Printstream();
```

Date :

Composition

- Composition is a strong from of association where contained object can not exist without the container.

Aggregation

Aggregation is a weak from of association in which contained object can exist without a container.

16/10/24

Method Binding

→ method binding it is the process of ~~joing~~ joining the method declaration with method diffinition.

→ the method binding activity done by the JVM at run time. when the method ~~its~~ gets called

<access-modifier><modifer> return type $\boxed{\text{methodname(<argument>)}}$

{ } → method definition

method signature.

Polymorphism

- polymorphism is the concept of one object showing different behaviors in it life time

→ In Java we have two types of polymorphism

Date :

(1) Compile time polymorphism
(2) Run time polymorphism

(1) Compile time polymorphism

method overloading

Ex①

```
class flipkart{
    void payment(){
        sopln("cod");
    }
    void payment(long cored){
        sopln("cored: 20% Discount");
    }
    void payment(String upi){
        sopln("upi: upto 100rs cashback");
    }
}

    mainclass1{
    ps v m(-){
flipkart flip=new flipkart();
flip.payment();
flip.payment(2345678910821);
flip.payment("abc@123");
    }
}
```

Date :

Ex 2.

```
Class Irctc{
    void search(int trainNumber){
        Sopln("Train schedeule");
    }
    Void search(String src, String dest){
        Sopln("List of Trains");
    }
    void search(String pnr){
        Sopln("Ticket Details");
    }
}

public mainclass2{
    psvm(-){

Irctc obj=new Irctc();
obj.search(26081);
obj.search("KSR","bpl");
obj.search("7898346811");
    }
}
```

Ex 3

```
Class Whatsapp{
    void send(String s){
        Sopln("Text Message");
    }
    Void send(Audio a){
        Sopln("Text Audio message");
    }
    void send(Image i){
        Sopln("Image message");
    }
```

Date :

```
Void send(Video v){
  Sopln("Video message");
}
}
public class mainclass3{
  ps v m(--){
Whatsapp ref=new Whatsapp();
 ref.send("Hello");
 ref.send(new Audio());
 ref.send(new Image());
 ref.send(new video());
}
}
```

Dummy.Java

```
class Audio{

}
class Image{

}

class video{

}
```

Date :

Ex 4

```
class TestScript {
    void test(int a){
    Sopln("Ad-Hoc - testing");
    }
    void test(double d) {
      Sopln("sanity testing");
    }
    void test(String s ){
     sopln("Regression testing");
    }
}

public class mainclass4{
    ps v m(-){
Testscript ta=new TestScript();
ta.test(45);
ta.test(4.5);
ta.test("ABC");
}
}
```

(1) We can do Compile time polymorphism using method overeloceding.

(2) In Compile time polymorphism the method binding decision are taken by the Compilere. at Compile time. based on the argument

(3) Decission taken by the compiler can not be changed hence it is also known as Static polymorphism or Static binding.

Date :

(4) The method binding decission are taken at the compile time before program start the execution. hence it is known as eavely binding.

17/10/24

Run Time Polymorphism

* overriding

```
class Facebook
{
 void display()
{
  soptln(" Display Response");
}
}
class FacebookWeb extends Facebook
{
  @overreide
   void display()
{
  Sopln("Display in Browser");
}}
class FacebookMobile extends Facebook
   @ overreide
    void display()
{
  Sopln("display in Mobile");
}
public class meinclass{
   p s v m (—){
Facebook fb;
fb= new FacebookWeb();
fb.display();
```

```
fb = new facebookMobile();
fb. display();
}
}
```

Ex(2)

```
class Card
{
  void swipe()
  {
    Soptln("Please wait..");
  }
}
class DebitCard extends Card
    @override
    void swipe()
  {
    Sopth("Account Balance Reduces");
  }
}
class CreditCard extends Card
    @override
    void swipe()
  {
    Soptln("credit due increase");
  }
}
public class mainclass{
    p s v m (-){
  Card ref;
  ref = new DebitCard();
  ref. swipe();
  ref = new CreditCard();
  ref. swipe();
}
}
```

Date :

Run time Polymorphism

↳ We can do Run time polymorphism using method overeviding

↳ In Run time polymorphism the method binding decision are taken by the JVM at run time based on the current instance

↳ The Decision taken by the JVM can by changed hence it is also known as dynamic polymorphism or dynamic binding

↳ The method binding Decision are taken by the JVM at the run time during program execution hence it is known as late binding.

Difference beth CTP & RTP

Compile time polymorphism	Run-time polymorphism
(1) Achieved using method overloading	Achieved using Method overreiding.
(2) Method Binding Decision is taken by Compiler @ Compile table	Method Binding Decision is taken by JVM at runtime
(3) Binding Decision is based on arguments	Binding decision is based on current instance
(4) Binding Decisions cannot be changed hence it is static ploymorphism.	Binding decisions can be changed hence it is dynamic polymorphism.
Static	Dynamic

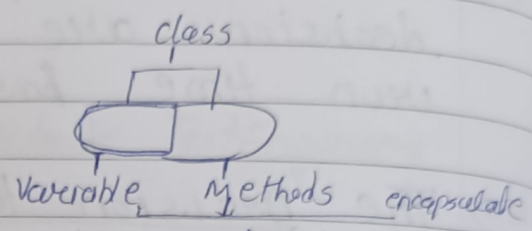
Date :

(5) Binding Decision are made before program starts executions hence it is called early binding

(5) Binding Decision are made during program execution hence it is known as late binding.

18/10/24

Encapsulation



→ Encapsulation is the concept of binding the realated the data member and member function into a class body

→ Every class in Java is an example for encapsulation because java syntax does not not allow the programmer to declare data member and member function out side a class.

Required-me

→ The main goal of encapsulation is data security

→ In java class is an encapsulation of data members and member function. a package is encapsulation of java programs

→ Jar file is an encapsulation of packages.

Under encapsulation

Access - Modifiers

- Access modifier can be used by the programmer to control the visibility of members inside a class.

- In java the following access modifier are available.

(i) private
(ii) <Default / Package>
(iii) protected
(iv) public

① Package com.jspiders.encapsulation;

public class Foxtrot
{
private static int a=10;
static int b=20;
protected static int c=30;
public static int d=40;

public static void main(String[] args){
Sopln (Foxtrot.a);
Sopln (Foxtrot.b);
Sopln (Foxtrot.c);
Sopln (Foxtrot.d);
}
}

Date :

② package com.jspider.encapsulation;

```
public class Geronimo {
public static void main(String[]args){
    Soptln (Foxtrot.b);
    Soptln (Foxtrot.c);
    Soptln Foxtrot.d);
  }
}
```

③ package com.jspiders.intro;
Import com.jspiders.encapsulation.Foxtrot;

```
public class Flower Extends Foxtrot {
public static void main(String[]args){
    Soptln (Foxtrot.c);
    Soptln (Foxtrot.d);
  }
}
```

Private

→ Members with private access modifier can be accessed only inside the same class.

<Default / Package>

→ Members that are declared without an access modifier are said to have default or package level access.

Date :

→ Members with default or package level access can be accessed in the same package

Protected

→ Member with protected access modifier can be accessed directly in the same package

→ Protected members can be accessed outside the package through inheritance.

Public

Members with Public access can be accessed directly ~~an~~from anywhere in the project

(4) package com.jspiders.encapsulation;
public class Karnataka {
  private static void test(){
    SopIn("executing test()---");
  }
  static void disp(){
    SopIn("executing disp()---");
  }
  protected static void play(){
    Sopln("executing play()--");
  }
  public static void send(){
    SopIn("executing send()--");
  }
  public static void main(String[] args){
    Karneetaka.test();
    Karnataka.disp();
    Karnataka.play();
    Karnataka.send();
  }
}

Date :

(2)
```
package com.jspiders.encapsulation;
public class TamilNadu {

    public static void main(String[] args) {
        Karnataka.disp();
        Karnataka.play();
        Karnataka.send();
    }
}
```

(3)
```
package come.jspiders.intro;
import com.jspiders.enscapsulation.Karnataka;
public class Bihar extends Karnataka {
public static void main(String[] args)
{
        Karnataka.play();
        Karnataka.send();
    }
}
```

Access modifier on a class / Interface / enum

→ a class / Interface / enum can only be declared with access modifier.

(1) public
(2) <default / package>

Date :

Ranking Table

Security	Visibility
1 private	1 public
2 <default/Package>	2 protected
3 protected	3 <default/Package>
4 public	4 private.

21/10/24

Overriding Rules on Access modifier

↳ Whenever the subclass overrides a method of the superclass it can re take the same access modifier or declare an access-modifier with higher visibility.

Super class	<default/package>	Protected	Public
	↑	↑	↑
Sub class	<default/package> protected public	protected public	public

Private
↑ CTE : we cannot inherit private methods

DATA HIDINGY

↳ It is the concept of hiding sensitive information from the outside world. by using private data member.

Ex ① Class Credentials
{
String username; // Not hidden
private String password; // Hidden
}

Ex ② Class Account
{
long accountNumber; // Not hidden
private double accountBalance; hidden
}

TIGHTLY ENCAPSULATION CLASS

↳ A Java class in which all data member are declared with private access modifier is called as tightly encapsulation class

eg:- Class Student
{
private long mobileNumber;
private String emailId;
}

eg:- Class Alpha
{
private int x;
private int y;
private int z;
}
Tightly encapsulation class

```
class Beta extends Alpha
{
    private int a;
    int b;
}
class Gamma extends Beta
{
    private int p;
    private int q;
}
```

Java Bean class / Pojo (Plain old Java object) class

→ Java Bean class can be used to achieve 100% encapsulation.

→ Java Bean class object are used to transfer data between programs.

→ Java Bean class object are called DTO (Data transfer object).

Rules to create a Java Bean class

1. All data members must be private.
2. Access to private data members via public getter and setter methods.
3. No methods except getter and setter methods
4. Java Bean class must have a default constructor.
5. must implement "Java.io.Serializable"

Date :

Import. Java. io. serializable

eg:-
```
public class Person implements serializable
{
    private int age;
    public int getAge()
{   // Authorization
        return age;
}
public void setAge (int age)
{   // Authorization.
    this. age= age;
}
}.

public class mainclass1
{
    Person p1 = new Person();
    p1 . setAge (16);
    System. out. println(P1. getAge());
}
}
```

Import java. io. serializable.

eg:-
```
public class Employee implements serializable
{
    private int eId;
    private double ctc;
    public int getEid()
{
        return eId;
}
    public void setEid (inteId)
{
    this. eId = eId;
}
```

```
public double getctc(){
    return ctc;
}
public void setctc(double ctc){
    this.ctc = ctc;
}
}
public class maeinclass2 {
public static void main(String[] args)
{   Employee ey=new Employee();
    ey.seteId(101);
    ey.setctc(4.5);
    sopln("EID:"+ey.geteId());
    sopln("CTC:"+ey.getctc());
}
}
```

```
import java.io.serializable
eg:- public class Account implements serializable
{
    private long accountNumber;
    private double accountBalance;
    public long getAccountNumber(){
        return accountNumber;
    }
    public void setAccountNumber(long accountNumber){
        this.accountNumber = accountNumber;
    }
    public double getAccountBalance(){
        return accountBalance;
    }
    public void setAccountBalance(double accountBalance){
        this.accountBalance = accountBalance;
    }
}
```

Date :

```
public class mainclass3 {
  p s v m(-){
Account acc = new Account();
acc.setAccountNumber(2345678910122L);
acc.setAccountBalance(25000.0);
Sopln("AccountNumber:"+acc.getAccountNumber());
Sopln("AccountBalance:"+acc.getAccountBalance());
}
}
```

Eg 1-

```
import java.io.serializable
public class Credential implements serializable
{
  private String UserName;
  private String password;
  public String getUserName()
  {
    return username;
  }
  public void setUserName(String username)
  {
    this.username = username;
  }
  public String getPassword()
  {
    return Password;
  }
  public void setPassword(String password)
  {
    this.password = password;
  }
}
```

Date :

```
public class Mainclass4
{
public static void main(String[] args)
{
    Credential c = new Credential();
    c.setUsername("admin");
    c.setPassword("password");
    Sopln(c.getUsername());
    Sopln(c.getPassword());
}
}
```

- Reader.java
- verify.java.

22/10/24

ABSTRACTION

→ In Greenreal abstraction is the concept of hiding Implementation Details and exposing functionality

→ In Java Abstraction is the concepts of hiding method Definition and exposing method Declaration.

→ In java we can do abstraction using
(1) Abstract Method
(2) Abstract class
(3) Interface

Abstract Method + Abstract class	— Partial Abstraction
Abstract method + Interface	— Pure / 100% Abstraction

Date :

Abstract Method

→ Abstract Method can we declaree using the keyword abstract

→ We can develop abstract method inside a abstract class or interface

→ Abstract method can not have a defination

→ Abstract method can not be static, final, and private

Abstract class

eg
```
abstract class Demo
{
  abstract void test();
  abstract void disp();
}
class Sample extends Demo
{
   @Overveide
   void test(){
   Sopln(" executing test()...");
  }
   @Overveide
   void disp(){
   Sopln(" executing disp()..");
}
}
```

Date :

```
public class Mainclass1 {
  p s v m (-) {
Demo  rey=new Sample();
  rey. test();
  rey. disp();
}
}
```

② abstract class Alpha

```
{
  abstract void play();
    void send() {
    sopln ("executing send()--");
  }
}
Class Beta extends Alpha
{
    @override
    void play() {
    sopln ("executing play()--");
  }
}
public class Mainclass2 {
  p s v m (-)
{
  Alpha rey = new Beta()) // upcasting
  rey. play();
  rey. send();
}
}
```

```
class Example extends Delta
{
}
public class Mainclass3
{
    P S V m(-)
Delta obj = new Example();// upcasting
 obj.run();
 ob.push();
}
}
```

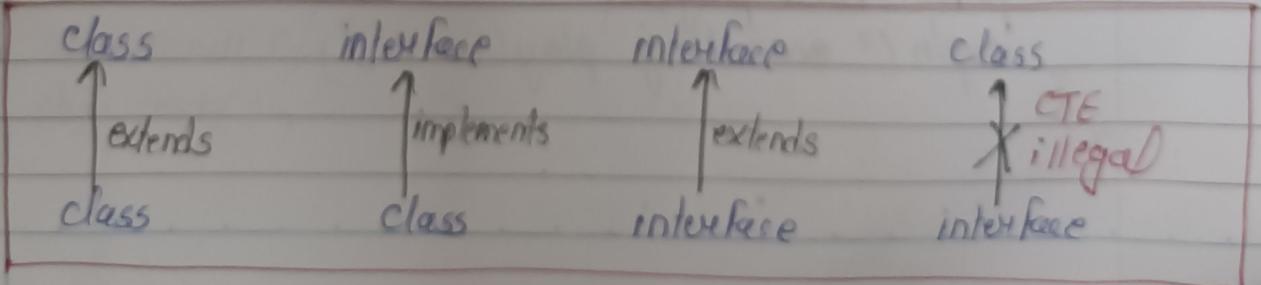
24/10/24

Abstract Class

① - Abstract class is created using abstract keyword
② - Inside an abstract class there is no restrictions on the data members and member functions
③ - We cannot create an object for abstract class.
④ - We can access the properties of an abstract class through inheritance and overriding
⑤ - Abstract class is considered as a partial abstract body.

Interface

interface InterfaceName
data member — public static final
member function — public abstract



Date :

① Package com.jspiders.pureabstraction;

interface Demo
{
    void test();
}

Class Sample implements Demo
{
    @Override
    public void test()
    {
        Sopln("Executing test()---");
    }
}

public class Mainclass1
{
    p s v m(-)
    {
        Demo rey = new Sample();
        rey.test();
    }
}

by default abstract

② package
interface Delta
{
    void play();
    void disp();
}

class Example implements Delta
{
    @Override
    public void disp()
    {
        Sopln("executing disp()---");
    }

    @Override
    public void play()
    {
        sopln("executing play()---");
    }
}

```
public class Mainclass2
public static void main(string [] args)
{
  Delta obj = new Example();
   obj. disp();
   obj. play();
}
}
```

③

```
interface Alpha {
      void test();
}
interface Beta {
      void send();
}
class Gamma implements Alpha, Beta{
      @Override
    public void send(){
  sopln(" excuting send()---");
}
  @Overrid
   public void test(){
  sopln (" excuting test()---");
}
}
public class Mainclass3{
  p s v m (—)
{
 Gamma obj=new Gamma();
 obj. send();
 obj. test();
}
}
```

(6)

```
interface Translator {
    void translate();
}
class KannadaTranslator implements Translator.
{
    @Override
    public void translate(){
    sopln("translate to Kannada");
}}

  class HindiTranslator implements Translator
{
    @Override
    public void translate(){
  Sopln("translate to Hindi");
}}

class FrenchTranslator implements Translator
{
    @Override
    public void translate(){
  Sopln("translate to French");
}}

class GoogleTranslate {
    static void translateTo(Translator t)
  {
    if(t!=null)
    {
      t.translate();
    }
  }
}

public class Mainclass6 {
    p s v m(-)
{
```

```
KannadaTranslator Kt = new KannadaTranslator();
HindiTranslator ht = new HindiTranslator();
FrenchTranslator ft = new FrenchTranslator();

GoogleTranslate.translateTo(Kt);
GoogleTranslate.translateTo(ht);
GoogleTranslate.translateTo(ft);
}
}
```

(7)

```
interface Driver
{
    void Connect();
}
class OracleDriver implements Driver
{
    @Override
    public void Connect(){
    Sop("Connected to oracle");
}}
class DB2Driver implements Driver
{
    @Override
    public void Connect(){
    Sop("Connected to DB2");
}}

class MicrosoftDriver implements Driver
{    @Override
    public void Connect(){
    Sop("Connected to microsoft SQL server");
}}
```

Date :

```
class DriverManager
{
  static void registerDriver(Driver reg)
  {
    if(reg != null)
    {
      reg.Connect();
    }
  }
}

public class MainClass {
  p s v m(-)
{
  OracleDriver reg = new OracleDriver();
  DriverManager.registerDriver(reg);

  MicrosoftDriver dr = new MicrosoftDriver();
  DriverManager.registerDriver(dr);
  }
}
```

25/10/24

Interface

(1) - An interface is declared using the keyword interface.

(2) - In inside an interface all data members are public, static and final.

(3) - Inside an interface all member functions are public and abstract.

(4) - We cannot create an object for an interface directly.

(5) - We can access properties of an interface by creating

an implementation class and overriding the abstract methods.

(6) - We can create an implementation class for an interface by using implements keyword.

(7) - A single class can provide implementation for multiple interfaces directly.

ABSTRACT CLASS	INTERFACE
(1) Created with abstract keyword	(1) Created with interface keyword
(2) no restriction on data member.	(2) all data members are public, static and final
(3) no restrictions on member functions	(3) all member functions are public and abstract
(4) We can inherit with extends keywords	(4) We can implement with implements keyword
(5) We cannot do multiple inheritance	(5) We can do multiple inheritance
(6) partial abstract body	(6) 100% or pure Abstract Body.
(7) will contain Constructor.	(7) will not have a Constructor.

Note - When a Concrete class provide implementation for two seperate interface. which is having method with same declaration it is enough to override once.

```
interface A.
{
  void test();
}
interface B
{
  void test();
}
```

```
class C implements A,B
{
  @Override
  public void test(){
  }
}
```

Date :

Note!- It is possible for class to simultaneously inherit from a class and an interface.

```
class A                    interface B
{                          {
}                          }
```

```
class C extends A implements B
{

}
```

Note :- When a class is inheriting from another class and interface of the same type if the super class contains the concrete method matching the abstract method in the interface it becomes optional to override the abstract method.

```
class A                    interface B
{                          {
public void run()            void run();
{                          }
}
}
```

concrete

```
class C extends A implements B
{


}
```

Date :

Functional Interface

↳ Any interface in Java that contains only one abstract method is called as a functional interface

↳ A functional interface should be necessarily preceded with the Annotation "@FunctionalInterface"

↳ Functional interface was previously known as SAM type interface. (Single abstract method)

Ex
```
@FunctionalInterface
interface playable{
    void play();
}
```

MARKER INTERFACE

no method body

↳ Marker interface are special type of interface which ~~has~~ contains an empty body.

↳ The implementation for this interfaces are provided internally by the JVM

Ex. java.lang.Cloneable
java.io.Serializable
java.util.RandomAccess
etc.

Java 8 Enhancement for interfaces

- In Java 8 to support functional programming several two major changes were made in the syntax of interface.

(i) interface supports static concrete methods
(ii) interface supports default concrete methods

```
interface Example
{
    static void play(){
    }
    default void test(){
    }
}
```

SINGLETON CLASS

- A java class for which there can exist only one object in the memory is called as a singleton class

Ex:-

```
Class Connection
{
    private static Connection con = null;
    public static Connection getConnection()
    {
        if (con == null){
            con = new Connection();
        }
        return con;
    }
```

(Connection: non-primitive data type; act as return type)

Date :

```
    private Connection(){
    }
}

public class maininclass1{
    p s v m(-){
Connection c1 = Connection.getConnection();
Connection c2 = Connection.getConnection();
Connection c3 = Connection.getConnection();
Connection c4 = Connection.getConnection();
}
```

Advantage of Singleton class

↳ Save same memory.
↳ enforces consistency in object of ~~behaviour~~ behaviours.

26/10/24

this Keyword

↳ This keyword is used to reffer to the ~~current~~ ~~connect~~ current of instance of the class
↳ This keyword can we used to access the non-static ~~data~~ members of the current class
↳ We can use this ~~to~~ ~~differentiate~~ keyword to differentiate between local variable and data members of current class.
↳ this keyword can only be used inside a non static context

* non-static context
  - method
  - IIB
  - Constructors

Date :

eg/-

```
Class Utility {
    void test() {
    sopln(this);
  }
}

public class Mainclass1 {
  public static void main(string [] args) {
    utility u1=new Utility();
    sopln(u1);
    u1.test();

  sopln("=====");
  Utility U2= new Utility();
  sopln(U2);
  U2.test();
}
}
```

eg/-

```
class Geronimo {
  ~~p s v m(-)~~ {
  int num=25;
  void test() {
    sopln("executing test()---");
  }

  void disp(){
    sopln(this.num);
  }  this.test();
}
public class mainclass2 {
  p s v m(-){
  Geronimo g=new Geronimo();
  g.disp();
}
}
```

o/p
25
executing test()---

Date :

eg-3:-

```
Class student
{
    String name;
    int id;
    student (String name, int id){
     this.name = name;
     this.id = id;
}}
class Hello {
    int a = 10;
    void test() {
    int a = 20;
   Sopln(a); //20
   Sopln(this.a); //10
}}
public class Mainclass3 {
   p s v m(-){
 student s1 = new student ("Ravi", 101);
 Hello h = new Hello();
  h.test();
}
}
```

~~eg-4~~

Super Keyword

(1) → Super keyword is used to access the non-static members of the super class

(2) → super keyword can only be return inside a non-static context.

eg - local variable

eg:1

```
Class Demo {
    int val = 25;
}
Class sample extends Demo {
    int val = 50;
    void play() {
        int val = 75;
        Sopln(super.val); // 25
        Sopln(this.val); // 50
        Sopln(val);      // 75
    }
}
public class Mainclass1 {
    p s v m(-) {
        Sample s = new sample();
        s.play();
    }
}
```

eg:2

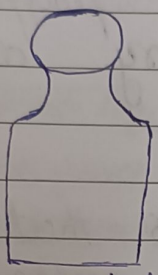
```
class softwarev1 {
    void login() {
        Sopln("Login with Username and Password");
    }
}
class softwarev2 extends softwarev1 {
    @Override
    void login() {
        super.login();
        Sopln("Captch validation");
    }
}
public class Mainclass2 {
    p s v m(-) {
        softwarev1 v1 = new softwarev1();
        v1.login();
        softwarev2 v2 = new softwarev2();
        v2.login();
    }
}
```

Date :

JAR File

Bytecode
.class

↳ Jar stands for java Archive
↳ Jar file is used to transport java programs
↳ the jar file contains compressed ."class" file
↳ All are file will have file extension ".jar"



28/10/24

module-3

Java Library

→ A Library is a group of predefined programs that will make programmers life easy

→ Libraries are classified into two types
(1) internal Libraries
(2) External Libraries.

Internal Libraries

↳ It is provided by the Languages creators
↳ It is provided as a part of the Language installer
↳ To make use of the program present in the internal Library. there is no need of additional Configuration
Ex. Core Java
C++ Libraries

External Libraries

The external Libraries are provided by third party vendors.

It is not a part of the language installer and has to be configured or downloaded separately.

To make use of the programs present in the external Library additional configuration is required.

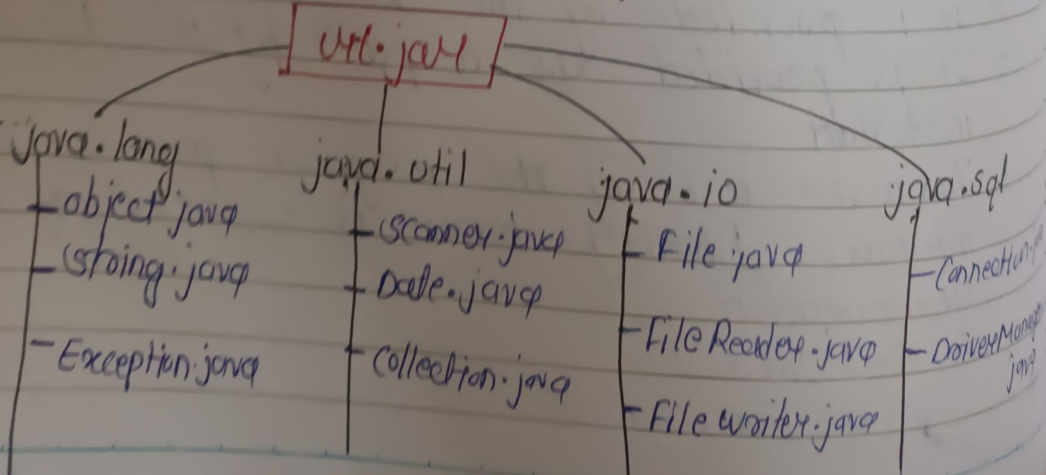
The external Libraries that we add to a projects build path is called as dependencies

rt.jar

It is jar file that is present in java Language that Contains all the inbuilt programs of Core java Library

The rt.jar file is organized in the form of packages

Each package Contains a Group of programs with similar functionality.

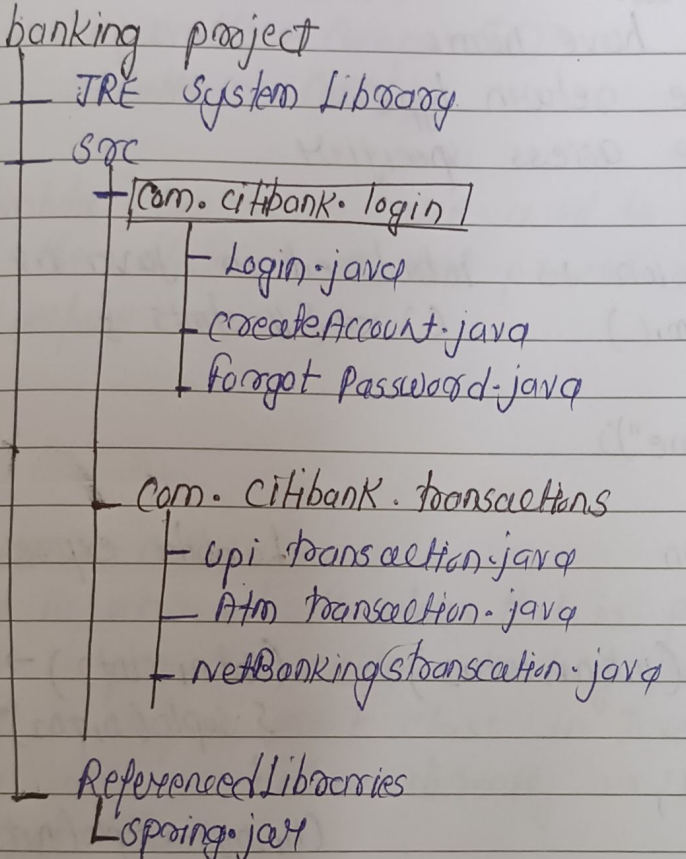


Date :

Package

~~A package~~ A package ~~it~~ is a location within a ~~It is~~ project or jar file which contains program having similar functionality.

- banking project
  - JRE System Library
  - src
    - com.citibank.login
      - Login.java
      - createAccount.java
      - forgot Password.java
    - com.citibank.transactions
      - upi transaction.java
      - Atm transaction.java
      - netBanking(transcation.java
  - ReferencedLibraries
    - spring.jar



Java.Lang Packages

→ It is a package in the core java Library that is present inside rt.jar file

→ It contains several programs that are related to Language fundamentals

→ The programs present in java.lang package can be directly accessed from any where in java programing

Date :

Lambda Expression

→ To introduced functional programming features
- Code optimization.
- Lambda expression is an Anonymous function
  Anonymous function is a function.

(1) which doesn't have name
(2) will not have return type
(3) will not have access specifier.

Lambda expression is introduced in java 1.8

```
public void m1()
{
  Sopln("Welcome");
}
```

() → { // stmts }

Normal function

```
public void sum(int n1, int n2)
{
 Sopln(n1+n2);
```

Lambda expression

```
(int n1, int2) →
{ Sopln(n1+n2);
}
(n1,n2)→ Sopln(n1+n2);
```

Note!-
It is not mandatory to specify the datatype for parameter if we do not specific Compiler will only add the datatype.

```
public int square(int n)
{
  return (n*n);
}
```

```
(int n) → { return (n*n); }
(n) → (n*n);
n → (n*n);
```

It is mandatory to write return stmts inside the curly braces.

Date :

Functional Interface

Any integer interface which are only one abstract method but can have any no of static and default methods

ex:- Runnable = run();
Comparable = compareto();
Callable = call();

- Lambda expression are used to give the implemention to the abstract method present in the functional interface.

4/11/24

OBJECT CLASS

- It is an inbuilt class that is present in Javalibrary
- It is present inside java.long package
- It is super most class in Java that is every class in java is either directly or indirectly a child of object class
- object class defines several common methods needed by all java programs.

Methods

- toString()
- hashcode()
- equals()
- wait(), wait(long), wait(long,int)
- notify()
- notify all()
- finalize()
- clone()
- getclass()

All method present in object class.

- tostring ek method hai jo har object ke sath hota hai
- ye method object class se inherit hota hai, object class java ki base class hai jo har class ka parent hota hai • iska default behavior yeh hai ki • tostring call

tostring()

public string tostring()

1) The default implementation of two string method provides string representation of the object

The string representation contains three elements
(1) FullyQualifield classname
(2) @.....
(3) Hexadecimal Pseudo Address

- if necessary the programmer can override to string method to return string value of their own choice.

- Whenever the programmer prints reference variable directly the JVM internally calls to string methods

Example

```
class student1 {
    string name = "Roma";
}
class student2 {
    string name = "Roma";
    @override
    public string tostring()
{
    return "student name:" + name;
}
}
public class mainclass1 {
    psvm (—) {
    student1 s1 = new student();
    sopln (s1);
```

Karte hai to yeh object ka class name aur uska hash code return karta hai

Date :

```
Student2 s2 = new Student2();
Sopln(s2);
}
}
```

hashCode()

public int hashcode()

(1) The default implementation of hascode method returns the unique integer id given by JVM.

(2) The unique integer id given by the JVM is decimal representation of the hexadecimal pseudo address.

(3) if necessary the programmer can override hascode method to return an integer value suited to their requirements.

Example:

```
class Employee1 {
  int employeeId = 101;
}

class Employee2 {
  int empolyeeId = 101;
  @overeride
  public int hashcode(){
    return employeeId)
}}

public class Mainclass3 {
psvm(-){
Employee e1 = new Employee1();
Sopln(e1.hashcode());
// Sopln(e1) internally call tostring method
Employee e2 = new Employee2();
Sopln(e2.hashcode());
}}
```

o/p 240650537
101

Date :

equals()

public boolean equals(object obj)

(1) The default implementation of equals() will work in the same way as "==" operator.

(2) If the programmer wants to do content comparision b/w object it is necessary to override equals() and provide require implementation.

Example:

```
circle class {
private int radius = 5;
@override
public boolean equals(object obj){
  Circle c = (circle) obj; // down casting.
  if (this = radius == c.radius){ internally upcasting.
      return true;
}
   else {
     return false;
   }
}

public class Mainclass4 {
  psvm (-){
  circle c1 = new circle();
  circle c2 = new circle();
  sopln (c1.equals(c2));
}
}
```

output

true

Example 2

```
class oneTimePassword
{
private int otp = 4545;
@Override
public boolean equals (object obj){
oneTimePassword otp = (oneTimePassword) obj;
if (this.otp == otp.otp){
    return true;
}
else {
    return false;
}
}
}
public class Mainclass5 {
    psvm (-){
oneTimePassword otp1 = new oneTimepassword();
oneTimePassword otp2 = new oneTimepassword();
Sopln (otp1.equals(otp2));
}
}
```

Example 3

```
class Account{
private long accountNumber = 350017410054L;
private long addharNumber = 123412341234L;
 @Override
public boolean equals (object obj)
{
Account acc = (Account) obj;
if (this.accountNumber == acc.accountNumber && this.
        addharNumber == acc.addharNumber)
{     return true;
}
```

```
    else
    {
        return false;
    }
}
}
public class Mainclass6{
  ps v m(-){
Account acc1=new Account();
Account acc2=new Account();
  sopln(acc1.equals(acc2));
}
}
```

5/11/24

whenever we override hashcode() of object class it is expected that will be also override the equals()

* wait(), wait(long), wait(long, int)

1. public final void wait()
2. public final void wait(long arg)
3. public final void wait(long arg1, int arg2)

The wait method is used to internally by the JVM to pause program execution till the occurrence of the event or for a certain time interval.

The wait method is overloaded in three versions
- All three versions of main method are final and cannot be overridden.

Date :

notify()

public final void notify()

- The JVM internally usage notify method to inform a particular thread about in occurance of an event

notifyAll()

public final void notifyAll()

- The JVM internally usages notifyAll() to inform all the threads about the occurence of an event

Garbage Collector

(i) It is a thread based mechanism that monitors object present in heap area.

- The garbage collector is responsible for destroying the unwanted objects present in heap area.
- Before the garbage collector destroys an object it will internally call the finalise method.
- The programmer can explicitly call the garbage collector by writing the java statements "System.gc()"

finalize()

protected void finalize() throws Throwable

- It is a protected method of object class that contains cleanup or closure code
- The code present in finalise method will be executed by the garbage collector before destroying an object

Date :

Example 1

```
class Person {
@overeveide
protected void Finalize() throws Throwable
{
  Sopln(" Tata ByeBye");
}
}
public class mainclass{
  psvm(-){
Person P = new Person();
System.gc()
Sopln("_____");
P = null;
System.gc();
}}
```

Example 2

```
public Connection
{
  @overeveide
  protected void Finalize() throws Throwable
{
  Sopln(" Close all resoweees");
}}
  public class mainclass{
    psvm(-){
Connection Con = new Connection();
System.gc();
Sopln("_____");
Con = null;
System.gc();
}
}
```

6/11/24 deep copy. shallow copy.

clone()

protected object clone() throws cloneNotSupported

- clone() of object class is used to perform object cloning i.e creating an exact replica of the existing object

points to Remember when using clone method

(1) clone() is risky and may cause "cloneNotsupportException"
(2) clone() can perform object clonning only when then object are cloneable type object
(3) whenever clone() performs object cloning the created copy will be internally ~~upcasting~~ upcasted to object class. hence it is mandatory to perform downcasting

Example 1

```
public class Game implements cloneable
{
  int hiscore;
public static void main(String[] args) throws
                        CloneNotSupportedException.
{
  Game obj = new Game();
  obj.hiscore = 100000;
Game copy = (Game) obj.clone();
  Sopln (copy.hiscore);
}
}
```

Date :

Example2

```
public class Square implements Cloneable
{
    int side;
    public static void main(String[] args) throws
                CloneNotSupportedException
    {
        Square original = new Square();
        original.side = 105;

        Square copy = (Square) original.clone();
        Sop(copy.side);
    }
}
```

Example3

```
public class Data implements Cloneable
{
    double size;
    psvm(--) throws CloneNotSupportedException
    {
        Data obj = new Dat();
        obj.size = 4.5;

        Data dey = (Data) obj.clone();
        Sop(dey.size);
    }
}
```

Cloneable type object

Any object that is created from a class which implements Cloneable marker interface is called as a cloneable type object

```
class Demo                          Class Demo implements cloneable
{                                   {
}                                   }
Demo obj = new Demo();              Demo obj = new Demo();
// NOT- CLONEABLE                   // cloneable.
```

NoTe!-
cloning in general is a restricted process and is discouraged.

getclass()

returntype

public final (class) getclass()

The getclass() method returns class class representation of the current instance that is the fully qualified classname

Example

```
package com.jspiders.examples;
import java.util.scanner;
class Demo
{
}
class sample extends Demo
{
}
public class mainclass1{
  ps vm(-){
object o1 = new Demo();
Sopln(o1.getclass());

object o2 = new Sample();
Sopln(o2.getclass());
```

Date :

```
object o3 = new scanner(system.in);
sopln(o3.getclass());

object o4 = new object();
sopln(o4.getclass());
}
}
```

	Compile time error	Exception	Runtime Error
identified by	Javac	JVM	JVM
Cause	syntax mistake	logical mistakes	System issues
overcome	code Rework	exception handling	Not a programmer issues
Examples	; expected cannot find symbol in compatible types	class CastException nullPointerException cloneNotSupported exception	vmoutey memo stackover linkage Error
	Error	Exception	

Exception Handling

7/11/24

- Exceptions are unexpected events that occur at the runtime and causes the program to terminat abnormally.

In Java exception are classified into two types

(I) checked exception

(II) unchecked exception.

Date :

Checked Exception
This exception are identify by the Compiler at Compile time

It is mandatory to handle checked exception
Example
CloneNotSupportedException
IOException
SQLException
etc ——

Unchecked Exception
This exception are not identified by the compiler at Compile time.

It is not mandatory to handle unchecked exception
Ex:-
ArithmeticException
nullPointerException
classCastException
etc ——

Note:
Both checked exception and unchecked exception will always occur at the runtime.

(8) Difference b/w checked exception and unchecked exception.

Checked exception	Unchecked exception
• identified by compiler at Compile time	• Not identified by compiler at compile time
• Mandatory to handle	• Not mandatory to handle

Date :

- Ex cloneNotsupportedException
  InterruptedException
  IoException
  etc

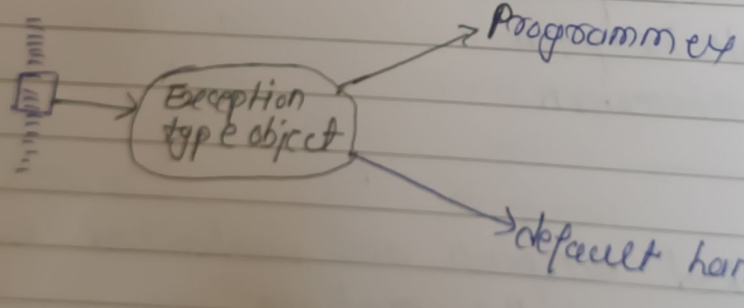
- ex: ArithmeticException
  nullpointerException
  classCastException
  etc.

note)
Whenever an exception occurs in the program the JVM internally creates an exception type object to represent the exception event

the exception object is created the JVM expected the programmer to handle the exception

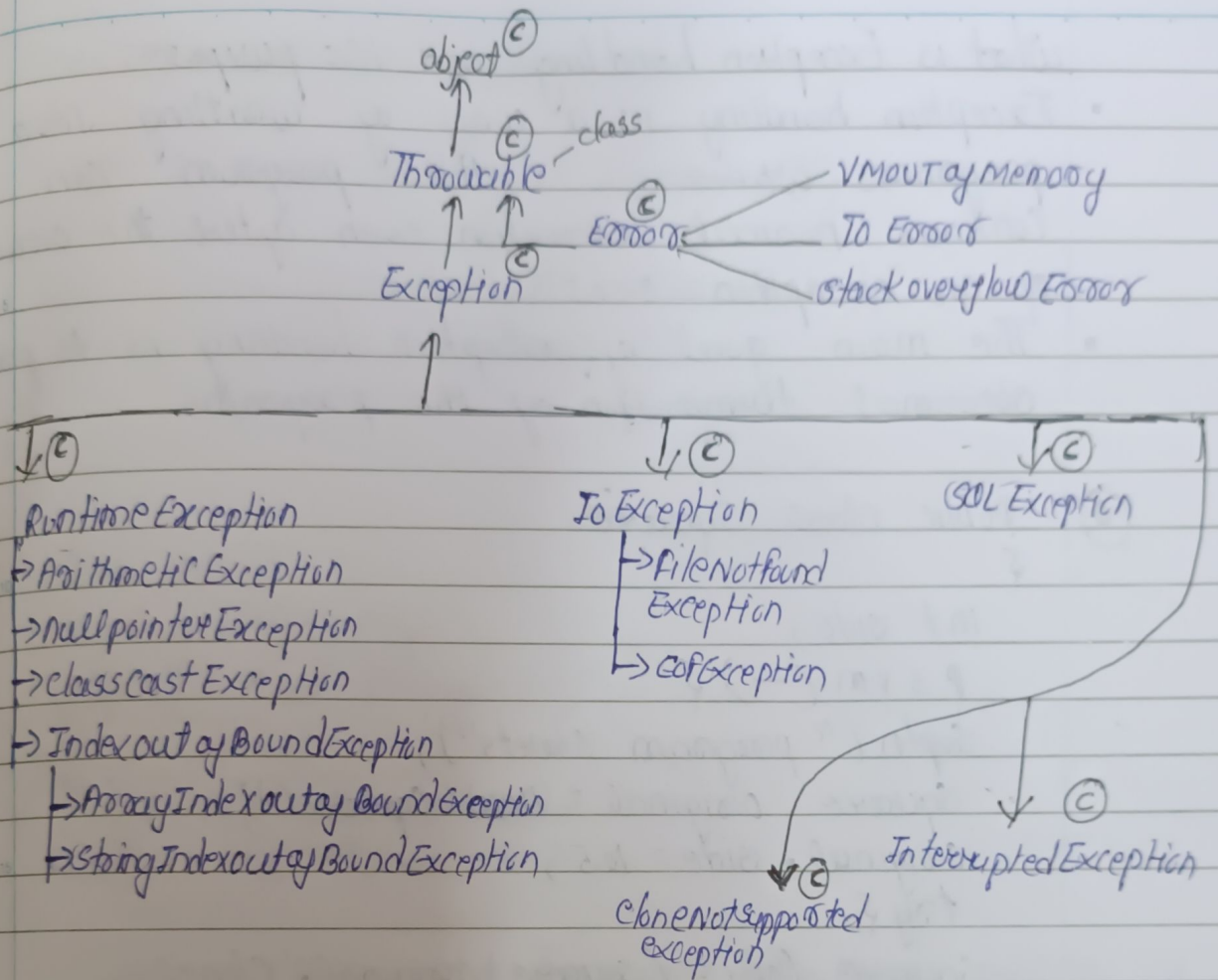
the programmer fails to handle the exception the JVM calls default handler.

class Example
{
3



Exception Hierarchy

- There is an inbuilt class in the java library for each and every exception



8/11/24

Exception Handling

(1)
```
Public class mainclass1 {
  PSVm(-){
 Sopln(" program starts");
   try
  {
   Sopln(10/0);
  }
  Catch (Arithmetic exception e)
  {
   Sopln("/By zero");
}
 Sopln(" program ends");
}
}
```

Date :

What is Exception handling and it's purpose

- Exception handling is a way of writting Java programs where the program can continue normal execution even after the occurance of an exception event
- The main goal of exception handling is to prevent abnormal termination of the programs.

(2)
```
public class Square
{
  int side;
  psvm(-){
  sopln("program starts");
  Square original = new Square();
  original.side = 105;
  try{
  Square Copy = (square) original.clone();
    sopln(Copy.side);
}
  catch(CloneNotSupportedException e)
{
  sopln("clone operation Failed");
}
  sopln("program ends");
}
}
```

(3)
```
public class mainclass2{
  psvm(-){
  String str = "jspiders";
  try
  {
  sopln(str.charAt(5));
}
```

```
catch (NullPointerException e)
{
 Sop ln ("Access with Null Reference");
}
Catch (StringIndexoutofBoundsException e){
 Sop ln ("Invalid Index Access");
}
}
}
```

(4)
```
class Hello{
 void greeting(){
 Sop ln("Hello Everyone");
}
}
public class mainclass {
   psvm(-){
   try{
  Hello h=(Hello) class.forName("com.jspiders.exceptions.Hello")
          ();
    h.greeting();
}
Catch (ClassNotFoundException e)
{
 Sop ln("No such class Exists");
}
Catch (IllegalAccessException e)
{
 Sop ln("Access Denied");
}
Catch (InstantiationException e){
   Sop ln("Instantiation Failed");
}
}
}
```

Date :

⑤
```
public class Maindass4{
  ps vm (-){
  try{
  sopln(" try starts");
  sopln(10/0);
  sopln("try ends");
  }
  Catch (ArithmeticException e){
    sopln("/By zero");
  }
  Finally
  sopln(" Bidyani");
}
}
```

⑥
```
Package.com.jspiders.exceptions;
import Java.util.InputMismatchException;
import Java.util.Scanner;
public class Mainclass5{
   ps vm(-){
Scanner scn=new Scanner(System.in);
  sopln(" Enter value");
   try{
    int val=scn.nextInt();
    sopln(" value:"+val);
}
  Catch(InputMismatchException e){
    sopln(" Invalid Input Entered");
}
  Finally
    scn.close();
    sopln(" scanner closed");
}
}
}
```

Exception Handling with try-catch and Finally.

try-block

- The try-block contains risky code (code that may cause exception)

- Every try block should be followed by atleast one catch block
- Control shifts from try-block to catch-block as soon as the exception occures
- A single try-block can be followed by multiple catch blocks.

Catch-block

- The catch-block contains recoverey code (code to be executed on occurrence of exception).
- Catch-block is written after the try-block
- Catch-block takes an object of exception type as argument
- Catch-block executes only when there is an exception event

Finally-block

- It is an optional block used in Exception Handling
- Code inside finally block will execute irrespective of exception.
- The finally block generally contains cleanup/-code

Date :

Different possible Combinations of try catch and finally

```
try
{
}
catch()
{
}
```

```
try
{
}
catch()
{
}
catch()
{
}
catch()
{
}
:
:
```

```
try
{
}
catch()
{
}
finally
{
}
```

```
try
{
}
finally
{
}
```

Allowed but not recommended

```
try
{
}
catch()
{
}
:
:
finally
{
}
```

```
try
{
}
finally
{
}
```

```
try
{
try{
}
catch(){
}
}
catch(){
try {
}
```

Date :

```
    Catch() {
  }
}
  finally {
    try {
  }
    catch() {
  }
}
```

9/11/24

Default Handler

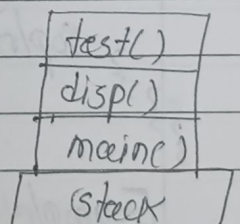
- The default Handler is an inbuilt mechanism that is called by the JVM
- When the programmer fails to handle and exception
- The default handler will perform the following task

1) Stop program execution
2) perform Stack Trace
3) Display the details
   - → Name of the Exception
   - → message (if available)
   - → Stack trace Info

Example 1

```
Package Com.jspiders.exceptions;
  public class Mainclass1 {
    static void test() {
    sopln ("test starts");
    sopln (10/0);
    sopln ("test ends");
}
  static void disp() {
  sopln ("disp starts");
  test();
```

test()
disp()
main()
Stack



main starts
disp starts
test starts.

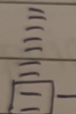
Date :

```
sopln(" disp ends");
}

ps vm (-){
sopln(" main starts");
disp();
sopln (" main ends");
}
}
```

throws Keyword

Example1



```
class Example
{
static void test() throws InterruptedException
{
    ⋮
    [-] → risky code InterruptedException
    ⋮
}
static void play()
{
  try {
  test();
}

catch( InterruptedException e){
  sopln(" ---- ");
}
}
}
```

Example1.2

```
public class Data
{
  double size;
  static void dataBackup() throws CloneNotSupportedException
{   Data obj = new Data();
    obj.size = 4.5;
```

```
Data key = (Data) obj.clone();
Sopln(key.size);
}
Public Static void main(String[] args)
{
  try{
  dataBackup();
  }
  Catch(cloneNotSupportedException

Example-3
class Hello {
  void greeting(){
  Sopln("Hello Everyone");
  }
}
public class mainclass3{
  Static void instantiate()
throws classNotFoundException, IllegalAccessException,
                                InstantiationException {
Hello h = (Hello)class.forName("com.jspiders.exceptions.Hello").
                                        ();
  h.greeting();
}
  Public Static void main (--){
    try{
    instantiate();
  } catch (classNotFoundException e){
    Sopln("No Such class found");
  } catch (IllegalAccessException e){
    Sopln("Access Denied");
  } Catch (InstantiationException e){
      Sopln("object creation Failed");
  }
}
}
```

Date :

throws Keyword

- throws keyword can be used by the programmer to propagate the responsibility of exception handling to the calling method
- throws keyword is always written in the method declaration.
- We can declare multiple exception using throws keyword

Note
throws keyword does not handle the exception it simply propagate the responsibility to calling method

Note
- checked exception can't be progagate by themselves hence programmer has to write throws keyword to support exception propagation

- Unchecked exception can propagate by themselves and the writting throws keyword optional.

```
public class Mainclass7
{
  public static void main(String[] args) throws Exception
  {
    /*
     * riskey code
     * execeptes
     *
     */
  }
}
```

Date :

In the above example main() method propagates the responsibility to JVM which handles the exception using default of handler.

PrintStackTrace(), getMessage()

- This are pre-defined methods declared in throwable class and this inheritant by all the classes in exception hierarchy.

- This method can be used by the programmer to run Exception Diagnostics.

ex:-

```
package com.jspiders.exception;

public class Mainclass1
{
  psvm(—)
  {
    try {
      sopln(10/0);
    }
    catch (ArithmeticException e)
    {
      String str = e.getMessage();
              sopln(str);
            e.printStackTrace();
    }
  }
}
```

Date : 11/11/24

Creating Custom Exceptions.

- It is possible for the programmer to declare and define custom exception in Java to suit their requirement

- To create a customis exception the programmer has to follow three steps

Step1 :- create a class that extends exception / RuntimeException
(checked / unchecked)

Step2 :- Develop trigger logic to cause the exception event

Step3 :- handle the exception

example :-

```
package com.jspiders.exceptions;
class CustomException extends Exception //1
{
}
class Trigger //2
{
static void verify(int num) throws CustomException
{
if(num%5==0)
throw new CustomException();
}
}

public class Mainclass //3
{
public static void main(String[] args)
{
```

```
try { //3

Trigger.verify(24);
Trigger.verify(30);
}
catch(customException e)
{
e.printstackTrace();
}
}
}
```

Example 2.

```
package com.jspiders.exceptions;
class overspeedException extends Exception{
}
class Highway patrol {
static void checkspeed(int speed) throws overspeedException
{
if (speed > 80) {
    throw new overspeedException();
}
}
}
public class mainclass9 {
    psvm(-){
    try {
    Highway patrol.checkspeed(68);
    Highway patrol.checkspeed(75);
    Highway patrol.checkspeed(120);
}
catch(overspeedException e)
{
e.printstackTrace();
}
}
}
```

Date :

Example3

```
package com.jspiders.exceptions;
class InvalidMobileNumberException extends Exception
{
}
class MobileNumberVerifier
{
static void verifyMobileNumber(String mobile) throws
                                InvalidMobileNumberException
{
  if (mobile.length()!=10){
     throw new InvalidMobileNumberException();
  }
}

  public class Mainclass10 {
     psvm (-){
     try
     {

  MobileNumberVerifier.verifyMobileNumber("6364889985");
  MobileNumberVerifier.verifyMobileNumber("8845633513");
}
    catch (InvalidMobileNumberException e){
      e.printStackTrace();
  }
}
}
```

Example 4

```
Package Com.jspiders.exceptions;
class LimitExceedException extends RuntimeException
{
}
class Atm {
 static int attempt = 0;
 static void checkAttempt()
{
   attempt ++;
   if (attempt > 5){
     throw new limitExceedException();
   }
}
}

public class Mainclass 11 {
     psvm (-) {
     try {
 Atm.checkAttempt();
 Atm.checkAttempt();
 Atm.checkAttempt();
 Atm.checkAttempt();
 Atm.checkAttempt();
}
catch (limitExceedException e){
        e.printStackTrace();
 }
}
}
```

Date :

Example 5

```
package com.jspiders.exception;
class InvalidIncidentLevelException extends Exception
{
}

class IncidentManager {
static void assignIncidentLevel(int level) throws
                    InvalidIncidentLevelException
{
  if (level>3) {
    throw new InvalidIncidentLevelException();
  }
}
}

public class MainclassIR {
   psvm (-) {
  try {
IncidentManager.assignIncidentLevel(2);
IncidentManager.assignIncidentLevel(1);
IncidentManager.assignIncidentLevel(3);
IncidentManager.assignIncidentLevel(5);
}
catch (InvalidIncidentLevelException e) {
        e.printStackTrace();
}
}
}
```

Date :

Example 6

```
package com.jspiders.exceptions;
class InvalidPincode Exception extends Exception
{
}
class Pincodeverifier
{
  class void verifyPincode (String Pin) throws
                        InvalidPincode Exception.
{
  if (Pin.length()!=6){
     throw new InvalidPincodeException();
}}
}
public class Mainclass11{
    psvm(-){
  try{
  pincodeverifier.verifyPincode("123456");
  Pincodeverifier.verifyPincode("560069");→not exception
  pincodeverifier.verifyPincode("489654");
}
catch (InvalidPincode Exception e){
        e.printStackTrace();
}
}
}
```

Date :

Generic Exception Handling

Note!

- In generic Exception handling a single catch block is written where the try block with object of class exception as an argument
- It is also possible to propagate multiple exception by writting throws exception in the method declaration.

The above approaches are not recommended.

(a) Differenace between throws and throw

throws	throw
• Written in method declaration	• Written in method Definition
• propagates exception to calling method	• Manually throw the exception object.
• Can propagate multiple exceptions.	• Can throw only one exception object

12/11/24

String class

- It is an inbuilt class present in "Java.lang" package
- String class is used by the programmer to represent a sequence of characters in the java program.

Date :

- it is a final class and can not have a subclass
- In java we can create string class objects in two ways.

(1) String literals
(2) String with new keywords

internally create objects String s1 = "Java";
String s2 = new String("Java

Example.

```
Package com.jspiders.Strings;
public class Mainclass1 {
    psvm (-) {
  String s1 = " jspider"; // Literal
    sopln( s1);
  String s2 = new String (" Ospiders"); // new Keyword
    sopln( s2);
  }
}
```

Note

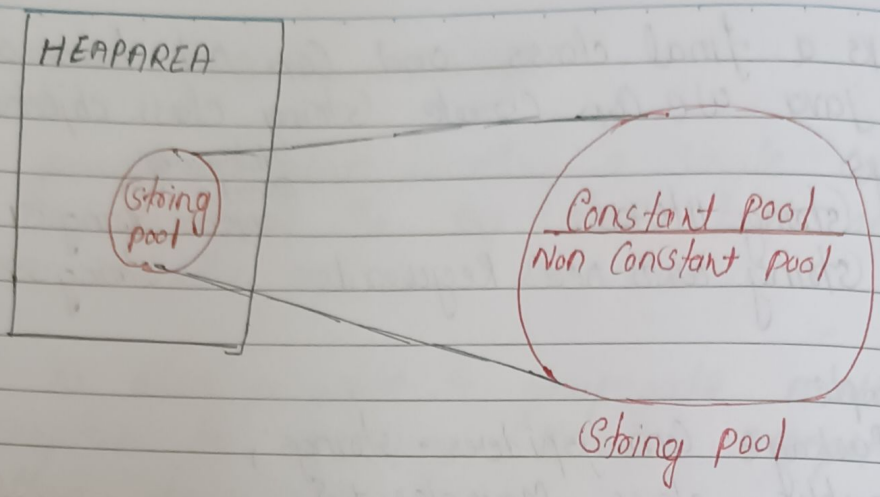
- String class object is created both when we use literal and new keyword approach.

Memory Management With respect to String Objects

- All String objects are allocated memory in a special part of a heap known as a string pool
- String pool is internally divided into two segments

① Constant pool
② Non constant Pool

Date :



Constant pool

- All String objects created as literal will be allocated memory in the Constant pool
- Before allocating memory in the Constant pool the JVM checks for duplicates.

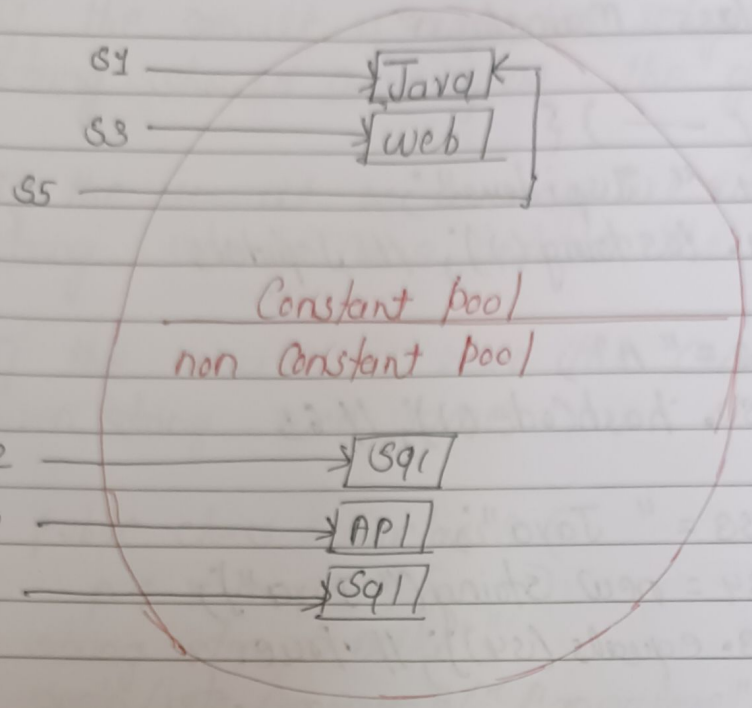
Non Constant pool

- All String objects created with new keyword are allocated memory in the non constant pool
- JVM does not check for duplicates before allocating memory in the non-Constant pool

```
String S1 = "Java";
String S2 = new String ("Sql");

String S3 = "web";
String S4 = new String ("API");

String S5 = "Java";
String S6 = new String ("Sql");
```



Overridden Methods of object Class In String

Note!-
String class overrides the following three methods of object class

toString()
hashCode()
equals()

1. The toString() is overridden to return the string value present in the current object

2. The hashCode() method is overriden to return UNICODE value of the current string characters

3. The equals() method is overriden to compare contents between 2 string objects.

```
public class mainclass{
{
    psvm(---){
    String s1 = "Jspiders";
    Sopln(s1.toString()); // Jspiders

    String s2 = "A";
    Sopln(s2.hashCode()); // 65

    String s3 = "Java";
    String s4 = new String("Java");
    Sopln(s3.equals(s4)); // true.
}}
```

Note:-
String classes inheriting all the methods of object class but overriding only toString(), hashCode() and equals()

String Class Objects are Comparable Type objects

- String Class implements the functional interface Comparable.
- String class provides implementation for the CompareTo() method present in Comparable Interface
- Using the CompareTo() method programmers can perform a lexicographical comparison between String object
- The CompareTo() method returns an integer value based on which decisions can be taken

Date :

-> If the result is a positive integer then Current String should come after the given String.

-> If the result is a negative integer then current String should come before the given String.

-> If the result is zero the Current String and given string are same

```
public class Mainclass3 {
    psvm (—) {
string str = "Delhi";
Sopln(str.CompareTo("Bangalore"); // +ve
Sopln(str.CompareTo("mumbai"); // -ve
Sopln(str.CompareTo("Delhi"); // 0
Sopln("Karna".CompareTo("Karnataka")); // -ve
Sopln("Bharatanatyam".CompareTo("Bharat")); // +ve
}
}
// ("Apple".CompareTo("apple"); -> -ve
    65                 97

[str].CompareTo[("-----")];
```

Current String

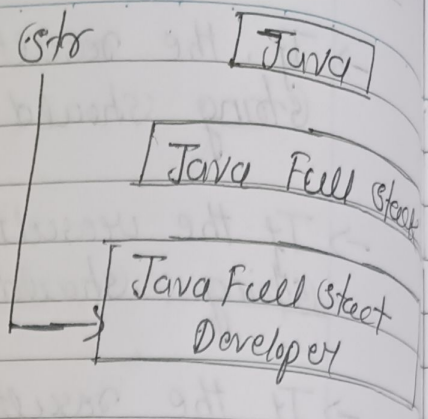
Given String

STRING CLASS OBJECTS ARE IMMUTABLE

- Once a String object is created it cannot be modified or changed. If the programmer tries to make changes or modifications it will internally create a new string object.

Date :

```
String str = "Java";
Sopln(str); → Java
str = str + " Full Stack";
Sopln(str); → Java Full Stack
str = str + " Developer";
Sopln(str); → Java Full Stack
                Developer.
```



- To overcome this drawback of writing class, Java provides StringBuffer and StringBuilder

(Q) What is Exception handling and Purpose?

- Exception handling is a way of writing java programs where the program can continue with normal execution even after the occurrence of an exception event.

- The main goal of exception Handling is to prevent abnormal termination of the program

13/11/24

STRINGBUFFER & STRINGBUILDER

- StringBuffer & StringBuilder are in-built classes present in "java.lang" package
- StringBuffer & StringBuilder can be used by the programmer as an Alternate for string class
- Both StringBuffer & StringBuilder are final classes
- We can create StringBuffer and StringBuilder using the new keyword
  (It doesn't support Literals)

```
public class mainclass4
{
    psvm(--){
    StringBuffer sb1 = new StringBuffer("Java");
    StringBuilder sb2 = new StringBuilder("python");
    sopln(sb1); // Java
    sopln(sb2); // python
    }
}
```

Note!!

- StringBuffer & StringBuilder both override only the "tostring()" method of object class

- The "tostring()" method is overriden to return the contents present inside the StringBuffer or StringBuilder object.

①
```
public class Mainclass5
{   psvm(--)
{   StringBuffer buffer = new StringBuffer("A");
    sopln(buffer.tostring()); // A
    sopln(buffer.hashcode()); // 41903949

    StringBuilder builder = new StringBuilder("a");
    sopln(builder.tostring()); // a
    sopln(builder.hashcode()); // 40897085
}
}
```

- StringBuffer & StringBuilder classes do not implement Comparable Interface hence, they are not Comparable type object.

Date :

- StringBuffer and StringBuilder object are mutable i.e it can be modified after creating.

Prog:-
```
public class mainclass6 {
    psvm (-) {
    String str = "sql";
    Sopln (str);
    str = str + "Database"; // Immutable: New object gets created internally
    Sopln (str);
    StringBuffer sb = new StringBuffer ("Java");
    Sopln (sb);
    sb.append ("full stack"); // mutable: existing objects gets modified
    Sopln (sb
    StringBuilder sb2 = new StringBuilder ("python");
    Sopln (sb2);
    sb2.append ("Data Science"); // mutable: existing object gets modified
    Sopln (sb2);
    }
}
```

append is (Concat) for StringBuffer and StringBuilder

STRING BUFFER
- Introduced in v1.0
- Legacy class
- Threadsafe and synchronized
- Secure
- Slow

STRING BUILDER
- Introduced in v1.5
- Non Legacy class
- Not Threadsafe and Not Synchronized
- less secure
- Fast

Date : _____

PARAMETER	STRING	STRINGBUFFER	STRINGBUILDER
- Version	V1.0	V1.0	V1.5
- Legacy class	Yes	Yes	No
- Literal creation	Yes	No	No
- Overriden methods of object class	tostring(), hashcode(), equals()	tostring()	tostring()
- Comparable Type	Yes	No	No
- Immutable	Yes	No	No
- Thread safe	Yes	Yes	No
- Synchronized	No	Yes	No
- Speed	Slow	Slow	Fast
- Secure	Secure	Secure	less secure

①
```
public class mainclass7{
 ps vm (--){
String str = "Tiger";
char ch[] = str.toCharArray(); // String --> char[]
sopln(ch); // Tiger

StringBuffer buffer = new StringBuffer(str); // String -> stringbuffer
sopln(buffer); // Tiger

StringBuilder builder = new StringBuilder(str); // String -> StringBuilder
sopln(builder); // Tiger
}
}
```

```
char ch[] = {'T', 'I', 'g', 'E', R'};
```

Date :

② public class mainclass7{
psvm(-){
char ch[] = {'J', 'a', 'v', 'a'};
String str = new String(ch); // char[] --> String
sop(str); // Java

StringBuffer buffer = new StringBuffer(str);
sop(buffer); // Java

StringBuilder builder = new StringBuilder(str);
sop(builder); // Java
}}

③ public class mainclass8{
psvm(-){
StringBuffer buffer = new StringBuffer("jdbc");
String str = buffer.toString(); // StringBuffer --> String
sop(str); // jdbc

char[] ch = str.toCharArray();
sop(ch); // jdbc

StringBuilder builder = new StringBuilder(str);
sop(builder); // jdbc
}}

④ public class mainclass9{
psvm(-){
StringBuilder builder = new StringBuilder("selenium");
String str = builder.toString(); // StringBuilder --> String
sop(str); // selenium

char[] ch = str.toCharArray();
sop(ch); // selenium

```
StringBuffer buffer = new StringBuffer(str);
sop(buffer); // selenium
}
}
```

WRAPPER CLASSES

- Wrapper classes are in-built classes in "java.lang" package.
- Wrapper classes are used to support Primitive Data Types
- All wrapper classes in java. are final & Immutable

Primitive Data Type	Wrapper class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

- Using Wrapper class we can perform following 3 operation

(1) Boxing
(2) Unboxing
(3) Passing.

(1) Boxing

- It is a Concept of representing a primitive wrapper class object value as a wrapper class object

Date :

```
int a = 10;
Integer obj = new Integer(a); // Boxing: int → Integer
                (deprecated)
Sopln(obj); // 10
double b = 4.5;
Double obj = b; // Boxing: double → Double
Sopln(obj); // 4.5

char c = 'J';
Character ch = c; // Boxing: char → Character
Sopln(ch); // J
```

(ii) UNBOXING

- It is a process of representing a wrapper class object as a primitive value.

```
Integer obj = 10;
int x = obj; // Unboxing: Integer → int
Sopln(x); // 10

Double val = 8.9;
double y = val; // Unboxing: Double → double
Sopln(y); // 8.9

Boolean bool = true;
boolean b = bool; // Unboxing Boolean → boolean
Sopln(b); // true.
```

14/11/24

PARSING

- It is the process of extracting primitive values that are embedded inside a string.

Date :

- To support passing the wrapper provides pre-defined parser method.

Wrapper class	Parser Method
Byte	parseByte()
Short	parseShort()
Integer	pareInt()
Long	pareLong()
Float	parseFloat()
Double	parseDouble()
character	NA
Boolean	parseBoolean()

①
```
package com.jspiders.wrapper;
public class Mainclass1{
    ps v m (—){
String s1 = "143";
int x = Integer.parseInt(s1);
Sopln(x); //143

String s2 = "6.89";
double y = Double.parseDouble(s2);
Sopln(y); //6.89

String s3 = "false";
boolean b = Boolean.parseBoolean(s3);
Sopln(b); // false.
}
}
```

- If we use an invalid parser method, it will cause a "NumberFormatException"

```
String str = "R-67";
int val = Integer.parseInt(str); // Exception → NumberFormat Exception.
Sopln(val);
```

Date :

- We can extract character values from a string using charAt() or toCharArray(). Hence, there is no need of a parser method.

```
String str = "admin";
char ch = str.charAt(4);
char[] ch = str.toCharArray();
```

eg:-
```
class Demo
{
  static void test(int a)
  { sopln("Kohli");
  }
  static void test(Integer a)
  { sopln("Rohit");
  }
  psvm(--)
  { Demo.test(10); // Kohli
  }
}
```

- Whenever there is a conflict between a primitive and Wrapper class, the conflict is resolved in favour of the primitive.

```
Object obj = null;
String str = null;

class Demo {
static void test(object obj)
{ sopln("Kohli");
}
static void test(String str)
{ sopln("Rohit");
}

public class Mainclass{
  psvm(--){
Demo.test(null); //Rohit
}
}
```

Date :

→ whenever their is a Conflict b/w Parent and child the ~~resoul~~ resolution is in favour of the child

```
Class Container
{   void insert (object obj)
    {
      sopln (" Inserted ");
    }
}
public class mainclass 5 {
    psvm (—) {
    Container c = new Container();
    c.insert (new object ());
    c.insert ("Java"); // UPCASTING: String ⟶ object
    c.insert (new StringBuffer ("SQL"); // upcasting: StringBuffer ⟶ object
    c.insert (25); // 1) Boxing: int ⟶ Integer
                      2) Upcasting: Integer ⟶ object
```

THREADS

why
How to create

- A Thread is a light-weight process i.e a program or application that is currently under execution and consumes less amount of system resources (CPU/Ram)

- Threads Concept is the one and only way in which a programmer Can perform multi-tasking at a program level

- The main goal of multi-threading is to improve the speed of execution.

DEFAULT THREADS IN JAVA

- Whenever a java program begins execution the JVM internally creates the following 3 threads

Date :

① main thread
② thread scheduler
③ Garbage Collector
→ DAEMON THREAD → low priority background
→ not under the control of programmer

① main Thread
- The main thread is responsible for executing the code written by the programmer.

② Thread Scheduler
- Thread scheduler is responsible for manging the life cycle of all threads in java.

③ Garbage Collector
- The Garbage Collector is responsible for memory management in heap area

- Thread scheduler and Garbage Collector and Daemon thread
- Daemon threads are low priority background thread which are not under the control of programmer

(8) How to create threads in Java? 15/11/24
- As a java programmer we can create threads in 2 way.
① By extending thread class
② By implementing Runnable Interface

① By Extending Thread Class - (start() RUN())
P.① Create a class that extends "java.lang.thread".
② Override "run()" method inherited from thread class
["run()" method should contain code to be executed by the new thread.]

Date :

Create an object of Target class.
Call "start()" method to begin thread execution

program

```
package com.jspiders.threads;
class DemoThreads extends thread //1
{
  @override //2
  public void run()
{
  for(int a=1; a<=5; a++)
  {
    sopln(" demo thread -- ");
  }
}}
public class Mainclass1 {
    ps vm (--) {
System.opln (" program start");
DemoThread dt=new DemoThread(); //3
dt.start(); //4 → execution starts from here
for (int a=1; a<=5; a++)        (new thread is created and calling start)
{
  sopln(" main Thread");
}
  sopln(" program ends");
}
}
```

BY IMPLEMENTING RUNNABLE INTERFACE → [RUN()]

create a class that implements "java.lang.Runnable"
override "run() method derived from Runnable
[run() method derived should contain code to be executed by the new thread.]

Date :

(3) Create an object of Target class
(4) create object of thread class by providing the target class object
(5) Call "start()" method to begin thread execution

program ②

```
class sampleThread implements Runnable //1
{ @override
  public void run() //2
{
  for (int a=1; a<=5; a++)
  { sopln("sample Thread")';
}}
public class mainclass2 {
  ps v m(-) {
sopln("program start");
sampleThread st=new sampleThread(); //3
Thread th=new Thread(st); //4
th.start(); //5
for(int a=1; a<=5; a++)
{
  sopln("main thread-");
}
sopln("program ends");
}}
```

Note! 7;
boolean extending thread class & implementing Runnable interface the recommended approach for creating a thread is implementing Runnable interface as it provides / offers more options for inheritance.

Date :

- ROLE OF "start()" method
- An actual thread is created in java memory only when we call the "start()" method.
- The start() method performs the following 3 tasks.

(i) Registers Thread with JVM
(ii) Allocate resource needed for execution
(iii) internally calls "run()" method.

(Q) what happens if we call run() method directly instead of calling start() method?
(i) No new thread will be created in java memory
(ii) The code inside run() method will be executed as a part of current Thread.

Thread.currentThread()
- currentThread() is a static method present in Thread class
- we can use this method to get the currently executing thread as an object of Thread class

Syntax:
```
Thread th = Thread.currentThread();
public class Mainclass3
{ psvm (--){
  Sopln("program start");
  for(int a=1; a<=5; a++)
{ Thread th = Thread.currentThread();
  Sopln("ID:" th.getId()+"Name:"+th.getName()
  +"priority:"+th.getPriority());
}
  Sopln("program ends");
}
```

Date :

Thread Properties

(I) ID
(II) Name
(III) Priority.

- Every Thread created in java will have the following 8 properties.

(I) ID
- It is an integer value given by the JVM
- It is used by the JVM to keep track of the Thread

(II) Name
- It's a (string value which can be given either by the programmer or by the JVM

(III) Priority
- It is an integer value ranging b/w 1 to 10
- with 1 being the lowest priority & 10 being the highest priority (Arg or default priority-5)
- We can access the thread priorities by using the getter() methods provided in thread class
  (i) getId() (ii) getName() (iii) getPriority()
- We can assign thread priorities by calling the (setter() methods of thread class
  ii) setPriority(int) i) setName(String)

Note :-
(1) Id is a read-only property i.e we can access the Id value but we cannot assign.

2) The priority of a thread is checked only when there is a resource crunch.

Date: 18/11/24

SYNCHRONIZATION

- It is the concept of making and method / resources thread safe
- We can performed synchronization in java using the keyword synchronized.
- If a method or resources is declared with synchronized keyword it can only we accessed by 1 thread at a time

Example 1

read, write, modify

```
class Application
{
    void read()
    {
    }
    synchronized void write()
    {
    }
    void modify()
    {
        synchronized
        {
        }
    }
}
```

T2 T2 T3 T4 T5 → read

T1 T2 T3 T4 T5 → write

one by one

Example 2

```
class BookMyshow
{
    void browse()
    {
    }
}
```

Date :

```
synchronized void payment()
{

}

void selectseat()
{
  ≡
  synchronized
  {
    ≡
    :
  }
}
}
```

If all the method in a class is ~~created~~ declared with synchorized keyword then the class can be considered as a thread safe class

```
class Example
{
synchronized void test()
{

}
synchronized void disp()
{

}
synchronized void plg()
{

}
}
```

Date :

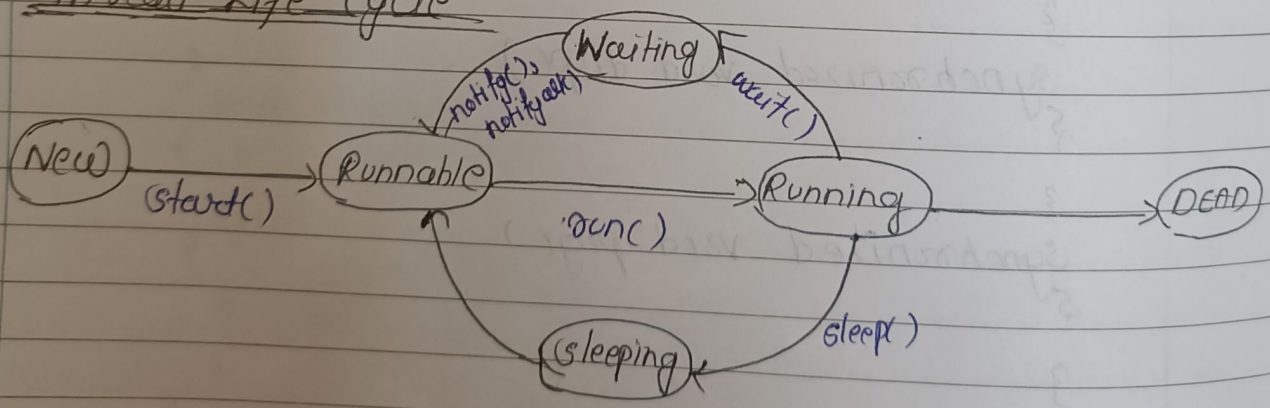
Thread.sleep()

- sleep is a static method present in thread class
- The programmer can use sleep method to pause program execution for a certain time interval.
- sleep method is risky and may cause interrupted exception

Example 1

```
package com.jspiders.threads;
public class MainClass5
{
ps v m (-) throws InterruptedException
{
    Sopln("program start");
    Thread.sleep(5000);
    Sopln("program ends");
}
}
```

Thread Life Cycle



- **New**
  - In this state thread is a newly created object.

Runnable
- In this state thread is ready to begin execution.

Date :

Running
- In this state thread is currently executing.

Dead Waiting
- In this state thread execution is paused by JVM

Sleeping
- In this state thread execution is paused by programmer

DEAD
- In this state thread has completed execution.

Race Condition

- Multiple threads competing for the same resource is called as race condition.

- Race Condition may lead to data inconsistency

- We can prevent Race Condition by using synchronization

Dead lock

In a Deadlock scenario two or more threads are mutually waiting for each other. to complete before starting execution.

- In a deadlock scenario the resource become idle and is not utilized.

- We can prevent deadlock scenario by using inter process thread communication.

Date :

(Q) What is a thread?
Define
use
How & create

(Q) Which is the recommended approach to create threads and why?
(Q) What happens if we call run() method directly instead of calling start() method?
(Q) What is synchronization?
(Q) Explain the working of start() method?
(Q) Explain the role of CurrentThread() method?
(Q) Explain the role of sleep() method?
(Q) List and explain thread properties?
(Q) Explain thread life cycle?
(Q) What are daemon threads?
(Q) What are the default threads in Java?

FILE HANDLING

File :-
- A file is the container ~~that~~ that hold the information and has a well known file extension.

Folder / Directory :-
A fold / directory is a container that contains files

File system :-
It is an arrangement of files and folder in a computer.

Date :

File handling

- It is the concept of manipulating the files and folder present in the a computer.

- Java library provides several pre-defined programs which can be used by the programmer to perform file handling.

- All pre-defined java program related to file handling can be found inside "java.io" package

Note:- The Java file handling concept is based on unix operating system.

19/11/24

Create a folder/Directory

```
import java.io.File;
// create a folder/directory.
public class Mainclass{
    psvm(--){
File obj = new File("D:/FILEIO");
// File data type is present in "io.File package"
here, we are giving = DriveName: foldername
boolean flag = obj.exists();
if(flag == false){
obj.mkdir(); // To create folder if not present
sop("folder created");
}
else{
    sop("folder already exists");
}
}
```

Date :

Create a file

```
import java.io.File;
import java.io.IOException;

public class mainclass{
    psvm(-){
File obj = new File("D:/FILEIO/Demo.txt");
 boolean flag = obj.exists();
 if( flag == false){
    try{
      obj.createNewFile();
      Sopln("file created");
    }
 Catch (IOException e){
    e.printStackTrace();
 }
 }else{
    Sopln("file already created");
 }
}
```

Delete a folder/file

```
import java.io.File;
public class mainclass{
    psvm(-){
File obj = new File("D:/FILEIO/demo.txt");
 boolean flag = obj.exists();
 if(flag == true){
    obj.delete();
 Sopln("Deleted");
}
```

Date :

```
else
{
  sopln("file folder not found");
}
}
}
```

- Java program delete the file/folder permanantly
- if folder is not empty it will not be deleted.

eg:-
```
import java.io.File;
public class mainclass {
  ps v m(-){
File rey = new File("D:/FILEIO/Demo.txt");
  boolean flag = rey.exists();
  if (flag == true) {
  sopln( rey.length());
  sopln( rey.getAbsolutePath());
}
else {
  sopln("file does not exists");
}
}
}
```

list()

```
import java.io.File;
public class mainclass{
  ps v m(-){
File rey = new File("D:/FILEIO");
String[]a = rey.list(); // It store all the files folder present
for(int i=0; i<a.length; i++){   in that folder.
  sopln(a[i]);
}
}
}
```

Date :

⇒ isFile() / isDirectory() // To check that is file or folder.
↑ for File ↑ for Folder

① import java.io.File;
public class mainclass {
ps v m (-) {
File dey = new File("D:/FILEIO");
String[] a = dey.list();
for (int i=0; i<a.length; i++) {
File f = new File(a[i]);
boolean flag = f.isFile(); // flag = f.isDirectory()
if (flag == true) {
sopln(a[i]);
}
}
}

Java.io.file / Java.io.File
- It is an inbuilt class present in java.io Package.
- The file class provides some common methods needed by all
- Java programs that perform basic file manipulation.

(i) exists()
(ii) mkdir()
(iii) createNewFile()
(iv) delete()
(v) length() — return type - long.
(vi) getAbsolutePath()
(vii) list()
(viii) isDirectory()
(ix) isFile()

Date :

- File ref = new File (string path);
File (File obj, string path);
File (string loc, string name);

we can make constructor as of three types

eg/ (i) File obj = new File(" C:/ ", " Gumman");
(ii) File ref = new File ("C:/ Gumman");
(iii) File f = new File(" C:/ ");
File ref = new (f, " Gumman");

#